Radix-64 Floating-Point Divider

Javier D. Bruguera
ARM Austin Design Center
Email: javier.bruguera@arm.com

***Abstract*—Digit-recurrence division is widely used in actual high-performance microprocessors because it presents a good trade-off in terms of performance, area and power. consumption. In this paper we present a radix-64 divider, providing 6 bits per cycle. To have an affordable implementation, each iteration is composed of three radix-4 iterations; speculation is used between consecutive radix-4 iterations to get a reduced timing. The result is a fast, low-latency floating-point divider, requiring 11, 6, and 4 cycles for double-precision, single-precision and half-precision floating-point division with normalized operands and result. One or two additional cycles are needed in case of subnormal operand(s) or result.**

I. Introduction

Division is one of the most representative floating-point functions in modern processors. There exist two main families of algorithms for calculating division in hardware [3]: *digit-recurrence algorithms*, which have linear convergence and are based on subtraction, and *multiplicative algorithms*, based on multiplication and with quadratic convergence. The energy efficiency of both approaches has been recently analyzed and the conclusion is that the digit-recurrence approach is much more energy efficiency [7] and requires less area.

In addition, for the floating-point precisions of interest, double, single and half-precision, digit-recurrence methods are much faster. Multiplicative methods rely on several iterations of a multiply-add fused (MAF) operation, and the latency of a single MAF is between 3 and 6 cycles [5], [9], [11]. In some cases, this is the latency of our proposed divider for single-precision.

In this paper a radix-64 digit-recurrence divider is described. It is hard to get an energy and timing efficient radix-64 implementation; then, three radix-4 iterations are overlapped in a single cycle providing 6 bits of the quotient per cycle, which is equivalent to a radix-64 iteration. In order to reduce the timing, speculation is used between consecutive radix-4 iterations in the cycle. The divider has been implemented in a processor with a frequency of 3 GHz.

Probably, the most critical point in digit-recurrence division is the quotient-digit selection. Every iteration, a digit of the quotient is obtained. To have a simple radix-4 selection function, independent of the divisor, the divisor needs to be scaled to value close enough to 1 [2]. This scaling is carried out before the digit iterations.

In addition, the first iteration, which gives the integer digit of the divider, with value $+1$ or $+2$, is carried out in parallel with the operands scaling, contributing to save one cycle in single-precision.

The result is a low-latency divider with 11, 6, and 4 cycles latency for double-precision, single-precision and half-precision, respectively, when the input operands and the result are normalized. These latencies include the scaling and rounding cycles. In case of subnormal operands, one or two additional normalization cycles are needed. Similarly, in case of subnormal result a second rounding cycle is needed.

The rest of the paper is organized as follows: In section II the main features of the proposed divider are outlined. Section III is a brief description of the foundations of digit-recurrence division. In section IV the detailed implementation of the divider is described. Finally, in Section V the divider is compared with other implementations in actual processors, and in Section VI the main conclusions are presented.

II. Main features

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 result is forced to be in $q \in [1, 2)$. Note that $q < 1$ only if $x < d$. This situation is detected in an early stage and the dividend if 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.

The algorithm used for the division is the radix-4 digit-recurrence algorithm with three iterations per cycle, 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.

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 to be scaled close to 1. Of course, to preserve the result the dividend needs to be scaled by the same amount than the divisor.

With the radix-4 algorithm, 2 bits of the quotient are obtained every iteration. As three radix-4 iterations are performed per clock cycle, 6 bits of the quotient are obtained every cycle, which is equivalent to a radix-64 divider.

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 prescaling, saving one additional iteration in single-precision.

On the other hand, there is an early-termination mode for exceptional operands. The early termination occurs when any of the operand are NaN, infinity, or zero, or in case of a division by a power of 2 with both operands normalized. In the latter case, the result is obtained by merely decrementing the exponent of the dividend.

In summary, the main features of the radix-64 divider are the following:

- Prescaling of divisor and dividend
- First quotient digit (integer digit) obtained in parallel with the operands prescaling
- Comparison of the scaled dividend and divisor and left shift of the dividend to have the result in $[1, 2)$
- Three radix-4 iterations per cycle, giving 6 bits per cycle
- Half, single and double-precision
- Subnormal support, with normalization cycles before the iterations
- Early termination for exceptional operands

III. Digit-recurrence division

Digit-recurrence division is an iterative algorithm which computes a radix-r quotient digit q_{i+1} and a remainder every iteration. The remainder $rem[i]$ is used to obtain the next radix-r digit. 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.

Particularizing to radix-4, $r = 4$, the partial quotient before iteration i is defined as

$$Q[i] = \sum_{j=0}^{i} q_j \times 4^{-j} \tag{1}$$

and the radix-4 algorithm, considering a scaled divisor close to 1, is described by the following equations,

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

$$rem[i+1] = 4 \times rem[i] - d \times q_{i+1} \tag{3}$$

being $\widehat{rem}[i]$ an estimation of the remainder $rem[i]$ with a few bits. For this implementation, it has been determined that only the 6 most-significant bits (MSB) of the remainder are required, three integer bits and three fractional bits [3].

Then, every iteration a quotient–digit is obtained from the current remainder, and a new remainder is computed for the next iteration. Then, the number of iterations is

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

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

The latency of the division, the number of cycles, is directly related to the number of iterations. It depends also on the number of iterations performed per cycle. Three iterations per cycle has been implemented to obtain 6 bits per cycle, which is equivalent to a radix-64 division. Then, the latency for a normal division is

$$cycles = \lceil it/3 \rceil + 2 \tag{5}$$

Apart from the cycles needed for the iterations, $\lceil it/3 \rceil$, there are two additional cycles for operand prescaling and rounding.

Some examples of digit-recurrence division, including radix-4, can be found in [1][3].

The naive implementation is shown in Figure 1. Note that only the most-significant bits of the remainder are used to select the quotient digit. The remainder is updated using carry-save adders (CSA) and stored in redundant representation. Then, the quotient digit selection needs the t most-significant bits of the remainder to be added in a carry-propagated adder (CPA) to get its non-redundant representation.

However, this naive implementation is too slow; to speed up the cycle, speculation in remainder calculation and quotient digit selection between iterations has been used, as explained in the following section.

IV. Architecture

The divider is composed of three parts: prescaling logic, digit-recurrence logic, and rounding logic. The prescaling and rounding take one cycle each, whereas the digit-recurrence logic, because of the iterative nature of the digit-recurrence algorithms, is reused during several consecutive cycles. In the following subsections the prescaling and digit logic are described. Rounding can be the standard digit-recurrence division rounding.

In addition, to reduce the latency by 1 cycle in single-precision, the first quotient digit, the integer digit which can take values $+1$ or $+2$ only, is calculated in parallel with the prescaling.

The number of bits n to be obtained in the iterations include the integer bit, the fractional bits, and the guard bit for rounding. If the integer digit is obtained in the operands prescaling cycle, the number of bits n is decremented by one. Then, for practical floating-point formats, the number of bits n, the number of iterations, and the number of cycles (see equations (4) and (5)) are:

- Double precision ($n = 53$): $it = 27$, $cycles = 11$
- Single precision ($n = 24$): $it = 12$, $cycles = 6$
- Half precision ($n = 11$): $it = 6$, $cycles = 4$

Note that in addition to the digit cycles, the latencies include the prescaling and rounding cycles.

Note that, due to the fact that the first quotient digit is calculated in parallel with the pre-scaling, in single precision the number of bits has been reduced from 25 to 24, and with $n = 25$ the number of iterations would be $it = 13$, and the

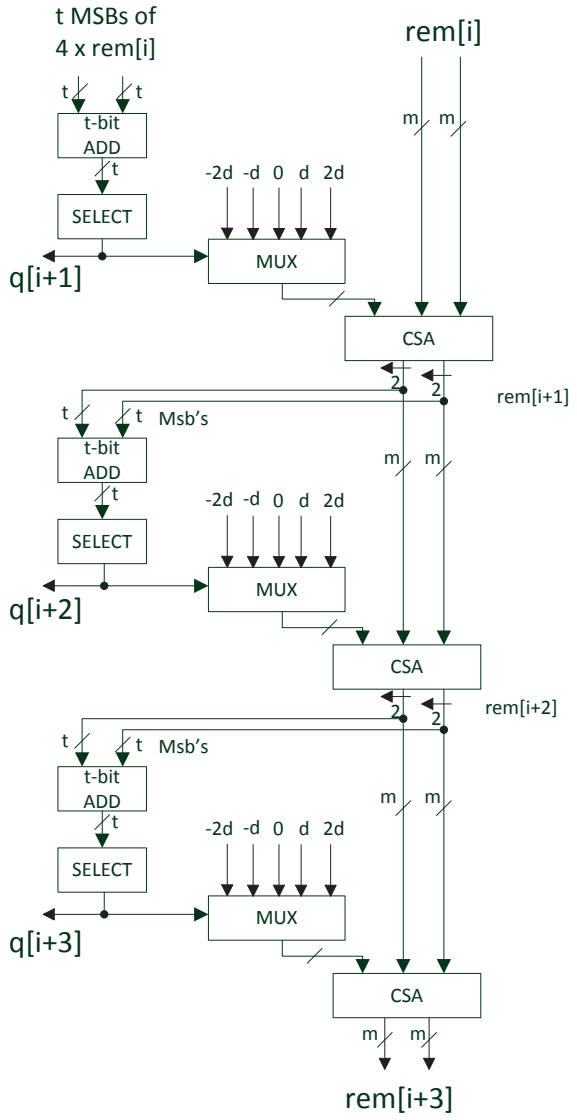

Fig. 1. Naive implementation of a radix-64 divider with three radix-4 iterations per cycle

number of cycles $cycles = 7$. Therefore, the latency has been reduced by 1 cycle.

A. Operand Prescaling and Integer Digit Calculation

During the prescaling, the divisor is scaled to a value close to 1 so that the quotient digit selection is independent of the divisor. It has been determined that, for a radix-4 digit recurrence, it is enough to have the scaled divisor in the range $[1 - 1/64, 1 + 1/8]$ [2].

The divisor is multiplied by a scaling factor $M = 1 + b \times 2^{-3}$, with $0 \leq b \leq 8$, and $b \neq 7$; this scaling factor depends only on the value of the divisor. As shown in Table I, only three bits of the divisor need to checked to get M. Note that for the prescaling, the divisor is supposed to be in $[0.5, 1)$. The prescaling has been implemented as the addition of the divisor plus 2 (or 1) multiples of the divisor [2]. The dividend has to be prescaled by the same amount to get the correct result.

TABLE I
DETERMINATION OF THE PRESCALING FACTOR

$0.1x_1x_2x_3$	M
000	1+1/2+1/2
001	1+1/4+1/2
010	1+1/2+1/8
011	1+1/2+0
100	1+1/4+1/8
101	1+1/4+0
110	1+0+1/8
111	1+0+1/8

The block diagram of this cycle is shown in Figure 2. During this cycle, in addition to the operands prescaling, the first iteration is carried out:

1) The operands are scaled. As part of the scaling, redundant carry-save representations of divisor and dividend are obtained.
2) The redundant prescaled divisor and dividend are assimilated to a non-redundant representation to get the remainder after the first iteration. The non-redundant divisor is used in the digit iterations as well.
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 prescaling.

In parallel with the operands' redundant to non-redundant conversion, the integer digit of the quotient is obtained as well. This is a simplified digit quotient calculation, because as the quotient is positive and in $[1, 2)$, the integer radix-4 digit can only take values $q_1 = +1$, or $q_1 = +2$. A simplified radix-4 iteration is performed to obtain the integer digit of the quotient. 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 and next remainder. Note that the difference between both cases is that the dividend is 1-bit left-shifted if the divisor is larger than the dividend.

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), shifted 1 bit to the left if the quotient digit is $+2$ and not-shifted if the quotient digit is $+1$.

B. 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. Note that, the implementation in

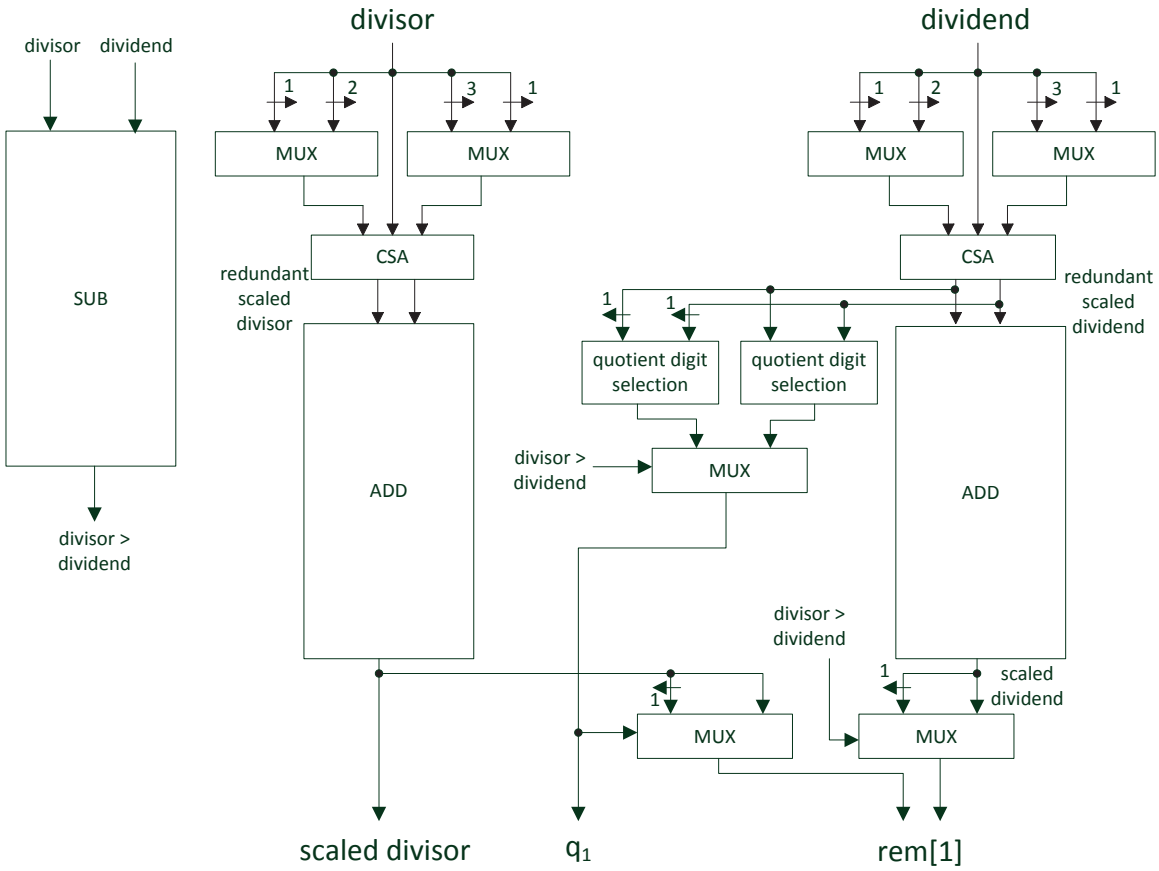

Fig. 2. Prescaling and integer quotient-digit calculation

Figure 3 is split into two parts, (1) digit selection and, (2) remainder calculation.

The remainders are computed speculatively according equation (3). So, five remainders are computed every iteration, one remainder for each value of the quotient 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.

The quotient-digit selection uses an estimation of the remainder to obtain the next quotient digit (equation (2)). As said in Section III, 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 II(a). The intervals $4 \times rem[i]$ for the selection of every digit has been obtained following the methodology described in [3].

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 another 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 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 II(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 II(b). Note that the selection of the interval end-points depends on the carry into the 6th MSB (*carry* column in the table) .

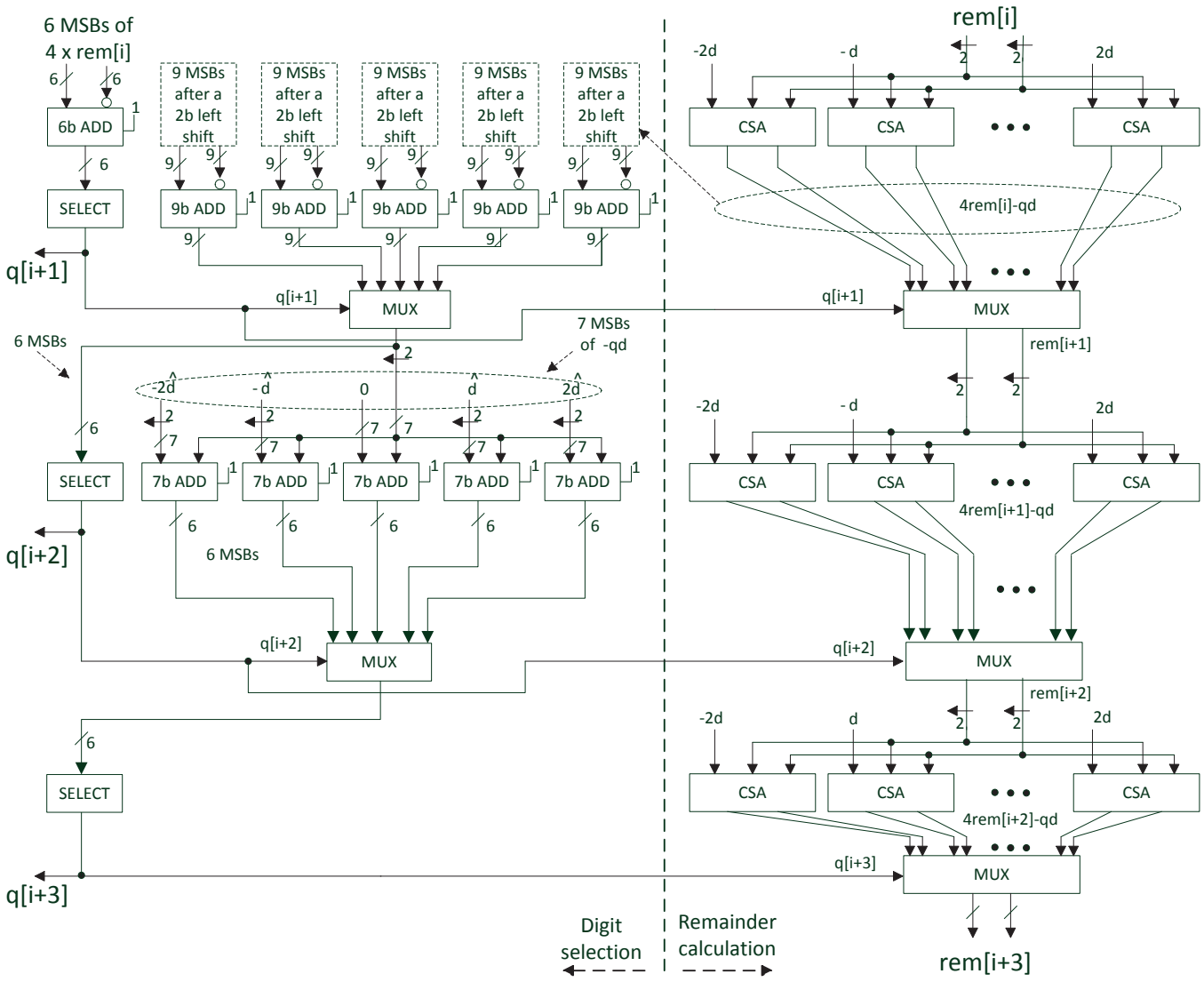

Fig. 3. Digit cycle logic

TABLE II
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

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 MBS 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 Table II(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.

The quotient is obtained in a redundant signed-digit representation with 2 words, a positive word ($quot\_pos$) storing the positive digits and a negative word ($quot\_neg$) storing the negative digits. For example, a final single precision quotient $quot = 1\ 0\ 0\ (-1)\ 2\ 1\ (-1)\ 0\ 0\ 0\ 1\ 1\ 1$ is represented by

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

Note that if the quotient digit is 0, there will be a 0 in both the positive and the negative words. The final quotient $Q[n]$ in equation (1) is obtained by subtracting both words and rounding the result. Alternatively, an *on-the-fly* conversion [1] could be used, but in our implementation this result in a worse cycle time.

V. EVALUATION

In this section we evaluate or design in terms of latency, area and timing ands compare it with other recent dividers.

A. Latency

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. This integer digit is obtained in parallel with the prescaling.

Hence, the number of digit cycles required for half, single and double precision are 2, 4 and 9 respectively. So, for normal operands the latency is

- Half precision, 4 cycles:
  E1/PS-DGT-DGT-RND1
- Single precision, 6 cycles:
  E1/PS-DGT-DGT-DGT-DGT-RND1
- Double precision, 11 cycles:
  E1/PS-DGT-DGT-DGT-DGT-DGT-DGT-DGT-DGT-DGT-RND1

being E1/PS the initial (where operands are unpacked and some conditions are determined), and prescaling cycle (scaling of divisor and dividend, left shift of the dividend if $x < d$ and integer digit calculation), DGT the digit cycle (3 radix-4 iterations per digit cycle), and RND1 the rounding cycle.

In case or normal operands, the initial stage and the prescaling stage are done in the same cycle. If any of the operands is subnormal, the initial and the prescaling stages are done in different cycles; this is because the operand needs to be normalized, that is in [1,2), before to be prescaled. So, in case of subnormal inputs there will be 1 or 2 additional normalization stages, NM1, NM2, before the prescaling (PS) cycle.

In case of a tiny result and additional rounding stage RND2, after RND1, is needed.

As an example, the latency of a single-precision division with one subnormal operand and tiny result is 9 cycles:

E1-NM1-PS-DGT-DGT-DGT-DGT-RND1-RND2 .

Table III compares the latency of the proposed divider with the latency of some other recent processors for floating-point half, single and double precisions with normalized operands and result [4], [6], [8] [10]. The latencies shown in the table

TABLE III
LATENCY COMPARISON

	HP	SP	DP
AMD K7 [8]	N/A	16	20
AMD Jaguar [10]	N/A	14	19
IBM z13 [4]	N/A	23	37
HAL Sparc [6]	N/A	16	19
This paper	4	6	11

include the iteration cycles and the pre- and post-processing cycles, such unpacking, prescaling and rounding. Note that no cycles for normalization are been included because it has been assumed that the operands are already normalized; although, as stated previously, the proposed divider can handle subnormal inputs and outputs.

Most of the design in the table uses a multiplicative division algorithm , [6], [8] [10], and one of them uses a radix-4 digit-recurrence implementation [4].

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 [4] uses a very low radix, which implies a high number of iterations. although its implementation is quite simple.

In our implementation we have been able to put in a single cycle three radix 4 iterations by using speculation between iteration in the same cycle. In addition, there are only two 2 pre- and post-processing cycles before the iterations, unpacking of operands and prescaling, and rounding.

B. Area

On the other hand, the divider area is larger in our implementation than in the other implementations in the table. Our divider uses a large number of CSAs and CPAs in the iterative part: five 58-bit CSAs for iteration for a total of 15 CSAs, five 9-bit CPAs, and 5 7-bit CPAs, plus the logic for the selection of 3 quotient digits and the multiplexers, twelve 58-bit 4:1 muxes and two 5:1 small wide muxes. In addition, in the prescaling logic three 58-bit adders and some additional logic, CSAs, multiplexers, and a reduced selection logic, are needed.

Multiplicative division algorithms 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 area of the radix-4 divider is also modest. 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 an stage of 3:2 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.

TABLE IV
DELAY OF BASIC GATES AND MODULES OF THE DIVIDER

		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
prescaling	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	9.34	56
	9-bit adder	11	66
	3:2 CSA	4	24
	5:1 mux	4.33	26
	SEL logic	$5.33 + \log_4(64)$*	50

*Due to fanout

The area of the rounding stage has not been included in the discussion because it should be roughly the same for all the implementations.

C. Timing

For the critical path delay estimation the Logical Effort model [12] is used in this section. Table IV 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 have be 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* module in the figure. This module consists of a 4 2-input logic levels, but the module output, the quotient digit, has a high fanout, roughly 64 gates.

Then, in the prescaling cycle there are two paths with roughly the path the same estimated delay,

$$\text{54-bit sub} \rightarrow \text{2:1 mux with large fanout} \rightarrow \text{2:1 mux}$$

and

$$\text{2:1 mux} \rightarrow \text{3:2 CSA} \rightarrow \text{58-bit adder} \rightarrow \text{2:1 mux}$$

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 is the one marked in blue in figure 3. It consists of

6-bit adder $\rightarrow$ *SEL* logic $\rightarrow$ 5:1 mux $\rightarrow$ SEL *logic* $\rightarrow$ 5:1 mux $\rightarrow$ *SEL* logic $\rightarrow$ 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 $300\ ps$.

VI. CONCLUSIONS

The architecture of a radix-64 floating-point divider providing 6 bits of the quotient per cycle is presented. To get a simple implementation and a affordable timing the radix-64 iteration is build with 3 radix-4 iterations, each one providing 2 bits of the quotient for a throughput of 6-bit per cycle, using speculation between consecutive iterations in the cycle.

Additionally, to have a simple digit selection logic, the divisor has been prescaled to a value close to 1, in such a way that the digit selection function does not depends on the divisor, it depends only on the 6 most-significant bits of the remainder. Prescaling 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 than the divisor as well.

Further latency reductions for some floating-point precisions 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 prescaling.

The result is a low latency floating-point digit-recurrence divider, with latencies of 11, 6 and 4 cycles for double-precision, single-precision and half-precision, respectively.

REFERENCES

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

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

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

[4] . 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).

[5] 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).

[6] 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).

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

[8] 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).

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

[10] 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).

[11] 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).

[12] 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) .