Simple Radix 2 Division and Square Root with Skipping of Some Addition Steps

Paolo Montuschi†

† Dip. di Automatica e Informatica,
Politecnico di Torino,
Corso Duca degli Abruzzi 24,
10129 Torino (Italy)

Luigi Ciminiera‡

‡ Dip. di Ingegneria Elettrica,
Università degli Studi di L'Aquila,
Poggio di Roio,
67040 L'Aquila (Italy)

Abstract

We present a new algorithm for shared radix 2 division and square root whose main characteristic is the ability to avoid any addition, when the digit 0 has been selected. Unlike other similar works, the solution presented uses a redundant representation of the partial remainder, while keeping the advantages of classical solutions. The paper shows how the next digit of the result can be selected even when the remainder is not updated, showing also the tradeoff arising. The average occurrences of 0 digit selections is also estimated in order to assess the benefits of the algorithm presented.

1 Introduction

Division and square root have always played primary roles in the computer arithmetic. Their importance has became more relevant with the recent introduction of the IEEE 754 floating point standard, as units complying with the IEEE 754 must include both division and square root in their instruction sets. Several architectures have been proposed for division, for square root and for shared division and square root in [11], [1] and more recently in [6], [4], [5], [10] and [2].

In this paper we present a new algorithm for shared radix 2 division and square root whose main characteristic is the possibility to avoid the addition required by *normal* steps when the digit 0 has been selected in the previous iteration. Actually, while *normal* iterations require that the partial remainder is updated by carrying out an addition/subtraction, *fast* iterations update the partial remainder simply with a left shift, with no addition/subtraction, and so can be executed faster.

The possibility to speed up the duration of the whole selection process by decreasing the number of additions, has been extensively explored in several works of the past literature. Robertson in [11] introduced the concept of redundancy with the precise aim of increasing the probability of selecting the digit 0, thus in effect reducing the number of additions necessary for updating the partial remainders. Wilson and Ledley in [14] claimed that with Robertson's method applied to division two digits per iteration were produced (on the average). The same paper [14] also cites a work by Smith and Weinberger [12], which concludes that for random sequences, the ratio of required cycles to quotient bits approaches 1/3 asymptotically. Freiman in [7] considers division and presents an analytical analysis based on Markov Chains, to determine the distributions of the remainders and of the divisors, for different ranges. In particular, in [7] it is shown that Robertson's algorithm produces 8/3 binary digits per iteration, a result which is confirmed by the simulation described in the same paper. Square root is considered in the detail by Metze in [8]. All the aforementioned methods refer to carry assimilated representations of the partial remainder, and then need long carry propagated additions, while a more attractive solution is certainly represented by partial remainders represented in redundant (e.g. carry save) form.

This paper studies again the problem of considering the selection of the digit 0, but from a different point of view. Our approach first considers the "classical" algorithm for shared radix 2 division and square root, with redundant representations of the partial remainders. Then, by introducing some modifications to the basic unit, an architecture is derived which could take the maximum advantage of the 0-selections, still maintaining the digit selection rules of the "classical" implementation. The simulation will provide tables reporting the average number of 0-selections in the case of different assimilations of the most significant bits of the partial remainders.

The paper is organized as follows: in section 2 we outline the proposed algorithm when applied to division, and we then provide the extension to square root in section 3. In section 4 our architecture for the proposed radix 2 division and square root algorithm is introduced. The evaluation is left to section 5, while the differences with previous works are discussed in section 6.

2 Division

The algorithm for computing the division x/d in base 2 [11], relies on the following formulae

$$w_i = 2(w_{i-1} - y_{i-1}d) = 2^{i+1}(x - Y_{i-1}d) \qquad (1)$$

CH3015-5/91/0000/0202$01.00 © 1991 IEEE

Table 1: Definitions of symbols used

i	iteration step
w_{i-1}	shifted partial remainder value at step $i-1$, with $w_0 = 2x$
y_i	digit of the partially developed result which has been computed at the i-th iteration of (1); $y_i \in \{-1, 0, +1\}$
d	divisor
x	dividend
Y_i	value of the quotient after the i-th iteration, with $Y_{-1} = 0$

$$Y_i = Y_{i-1} + 2^{-i} y_i \qquad (2)$$

where the symbols used are defined in Table 1. For the purposes of this paper we define an iteration as the set of operations between two consecutive digit selections. Therefore, iteration i is the set of operations occurring between the moment when y_{i-1} is selected and the moment when y_i is selected. For division, we consider x and d as being normalized to $1/2 \leq d, x < 1$. It turns out that the result $Y = x/d$ is normalized to $1/2 < Y < 2$. It is assumed for x and d to be available in their full precision and in carry assimilated form, at the beginning of the iterations, thus excluding from this analysis the on-line algorithms [13].

2.1 Previous work

Both the algorithm and the architecture for radix 2 division are well known from the existing literature [4] and Fig. 1 shows a sketch of the scheme of the "classical" architecture for radix 2 division. The shifted partial remainder w_{i-1} is stored in redundant form in the registers C and S. The carry lookahead adder CLA then operates the assimilation of the 4 most significant bits of w_{i-1} which is input to the digit selection table T. The selected digit y_{i-1} is then used to update the previous partial remainder by means of a carry save adder CSA and a shift of one position left (block SH). To handle conveniently the cases of positive and negative update of the partial remainder CSA can work in cooperation with a 1's complementor (COM). The on-the fly-conversion [3] of the partially developed result Y_i is carried out by the sub-unit OTFC.

When the shifted partial remainder is represented in carry save form [4], the selection rules for the digit y_{i-1} are:

$$\begin{aligned} &select \;\; y_{i-1} = +1 \quad if \quad 0 \leq \widehat{w}_{i-1} \leq 3/2 \\ &select \;\; y_{i-1} = 0 \quad if \quad \widehat{w}_{i-1} = -1/2 \\ &select \;\; y_{i-1} = -1 \quad if \quad -5/2 \leq \widehat{w}_{i-1} \leq -1 \end{aligned} \qquad (3)$$

where $\widehat{w}_{i-1}$ (expressed on 4 bits in two's complement) denotes the representation of w_{i-1} in carry assimilated form, truncated to the first fractional bit. It should be observed that the selection $y_{i-1} = +1$ (or $y_{i-1} = -1$) implies the necessity to update the shifted partial remainder by subtracting (adding) the divisor and by performing a left shift of one position. Conversely, when the selection is $y_{i-1} = 0$, it is only necessary to shift one position left w_{i-1} in order to get the new w_i.

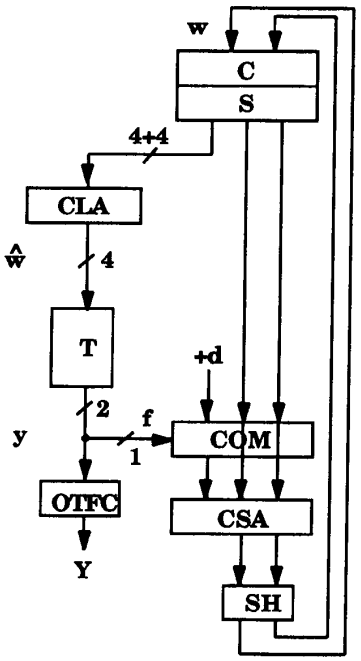

Figure 1: Conventional radix 2 division unit

2.2 The algorithm

The proposed algorithm comes from two consequences which arise when the digit $y_{i-1} = 0$ has been selected: *i)* to get the new shifted partial remainder w_i it is not necessary to pass through any carry save adder, but only to perform a left shift, and *ii)* the value of $\widehat{w}_i$ can be easily deduced from $\widehat{w}_{i-1}$; in fact, the information provided in output of the carry lookahead adder at iteration $i-1$ is also valid for iteration i since it is not affected by any manipulations other than a simple shift. These concepts can be better presented by referring to the numerical values. Let us assume the selection $y_{i-1} = 0$. From the selection rules (3), we have $\widehat{w}_{i-1} = -1/2$. Let us now consider the left shift required by (1) to get the new w_i from the previous w_{i-1}. With reference to Fig. 2, we can observe that it is possible to obtain the new value of $\widehat{w}_i$ from the previous $\widehat{w}_{i-1}$ in a very simple way. The three bits of $\widehat{w}_i$ from the weight 2^2 to the weight 2^0 are respectively the same as the three bits from the weight 2^1 to the weight 2^{-1} of $\widehat{w}_{i-1}$. This is due to the fact that when a 0 has been selected it is necessary to perform only sin-

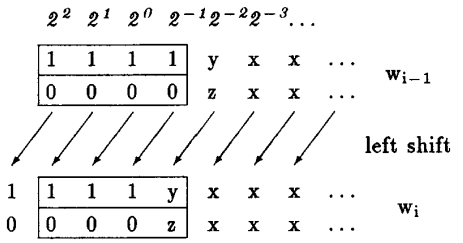

Figure 2: Updating of the partial remainder when the digit 0 has been selected

gle shift left, and in such a case the information given by the CLA is not corrupted. As can be seen in Fig. 2, the fourth bit (i.e. the least significant) which is necessary to determine completely the new $\widehat{w}_i$ used for the next digit selection, must come from the assimilation of the two bits of w_{i-1} with weight 2^{-2}, i.e. the two bits of w_i with weight 2^{-1}. Let us denote with a_i the assimilation of the two bits of weight 2^{-1} of w_i. Three possibilities exist:

1. The assimilation yields 0, which becomes the least significant bit of $\widehat{w}_i$. In fact, $\widehat{w}_{i-1} = (1111)_2$ implies $\widehat{w}_i = (1110)_2$, i.e. $\widehat{w}_i = 2\widehat{w}_{i-1} + 0 \cdot 2^{-1} = -1$. In such a case, according to the rules (3), the next digit selection must be $y_i = -1$.
2. The assimilation yields 1 and, according to (3), it must be $y_i = 0$.
3. The assimilation yields 2 and, according to (3), it must be $y_i = +1$.

Therefore, it should be noted that the digit selection at step i (when a 0 has been selected at step $i-1$), is very easy and it is only determined by the value of the assimilation of the two bits of w_{i-1} with weight 2^{-2}, because the information provided by output on the CLA is not corrupted by updates and can be reused in the selection procedure of the next iteration. Should a digit 0 be selected at step i too, the aforementioned process can still be applied for the selection procedure at step $i+1$, and so on. The proposed algorithm is formally expressed by the following selection rules:

- If $y_{i-1} \neq 0$

$$\begin{array}{llll} select & y_i = +1 & if & 0 \leq \widehat{w}_i \leq 3/2 \\ select & y_i = 0 & if & \widehat{w}_i = -1/2 \\ select & y_i = -1 & if & -5/2 \leq \widehat{w}_i \leq -1 \end{array} \quad (4)$$

- If $y_{i-1} = 0$ then

$$\begin{array}{llll} select & y_i = +1 & if & a_i = 2 \cdot 2^{-1} \\ select & y_i = 0 & if & a_i = 1 \cdot 2^{-1} \\ select & y_i = -1 & if & a_i = 0 \cdot 2^{-1} \end{array} \quad (5)$$

where a_i (expressed on 2 bits in binary form) denotes the result of the assimilation of the two bits of weight 2^{-1} of w_i.

3 Square root

The algorithm for square root, although different in the iteration step from the division recurrence (1), has been shown to present many similarities with division in the selection rules. This has allowed units to be designed where the same hardware is shared by division and square root [6]. The algorithm for computing the square root $\sqrt{x}$ in base 2 [10], relies on the following formulae

$$z_i = 2[z_{i-1} - (2Y_{i-1} + y_i 2^{-i})]y_i = 2(z_0 - Y_i^2)2^{i+1} \quad (6)$$

$$Y_i = Y_{i-1} + 2^{-i} y_i \quad (7)$$

where z_{i-1} is the shifted partial remainder value at step $i-1$, with $z_0 = 2x$, and the same notation introduced for division has been adopted. For square root we assume x as being normalized to $1/4 \leq x < 1$. It turns out that the result $Y = \sqrt{x}$ is normalized to $1/2 \leq Y < 1$. Also for square root, it is assumed for x to be available in its full precision and in carry assimilated form, at the beginning of the iterations. A key work in the study of binary square root is represented by [10], where an architecture which is very similar to the scheme of Fig. 1 is assumed as reference model, and where it is demonstrated that the selection rules are the same as for division (3), provided that the transformation of variables $\widehat{w}_{i-1} = \widehat{z}_{i-1}/2$ is taken into account, i.e. the fractional point is considered to be shifted by one position. It turns out that the same considerations expressed in section 2.2 are still valid, and also for square root our algorithm can be formally expressed in the same terms as the selection rules (4) and (5) with $\widehat{w}_{i-1} = \widehat{z}_{i-1}/2$.

4 Architecture for shared division and square root

4.1 Previous work

The selection rules for radix 2 division (3) need to be implemented by table T in Fig. 1. The quotient digit selection of y_i can be implemented in two ways [4], either by first using a carry lookahead adder (CLA) to assimilate the four most significant binary positions of w_i so as to produce $\widehat{w}_i$ and then using the four resulting bits as inputs to a combinatorial network for the selection, or by using the eight bits of the four most significant positions of the carry save representation of w_i as inputs to the combinatorial network. The main difference between the two choices is in the assimilation of the most significant part of w_i, which is performed by the carry lookahead adder CLA and by the selection table T itself, respectively. Therefore, the architecture sketched in Fig. 1 refers directly to an implementation

related to the first choice, while it is necessary to remove the CLA in order to obtain the architecture related to the second choice (actually, with a different and a (slightly) more complex selection table T).

Let us consider the combinational functions to implement the selection procedure [4]. We denote with $s_{i,3}s_{i,2}s_{i,1}s_{i,0}$ and $c_{i,3}c_{i,2}c_{i,1}c_{i,0}$ the sum and carry bits of the four most significant binary positions of w_i respectively, and with $r_{i,3}r_{i,2}r_{i,1}r_{i,0}$ their assimilation (i.e. the four most significant bits of $\hat{w}_i$) and we assume that a quotient digit represented in sign and magnitude (q_s and q_m, respectively) is produced, where $+1$ is represented by $(q_s, q_m) = (0, 1)$, 0 is represented by $(q_s, q_m) = (1, 0)$, and -1 is represented by $(q_s, q_m) = (1, 1)$. When a CLA is used, a possible pair of selection functions would be

$$\begin{aligned} q_s &= r_{i,3} \\ q_m &= \overline{r_{i,2}r_{i,1}r_{i,0}} \end{aligned} \tag{8}$$

Conversely, when the assimilation is performed by the selection functions themselves, according to [4] we have

$$\begin{aligned} q_s &= p_{i,3} \oplus (g_{i,2} + p_{i,2}g_{i,1} + p_{i,2}p_{i,1}g_{i,0}) \\ q_m &= \overline{p_{i,2}p_{i,1}p_{i,0}} \end{aligned} \tag{9}$$

where

$$p_{i,j} = s_{i,j} \oplus c_{i,j} \quad and \quad g_{i,j} = s_{i,j}c_{i,j} \tag{10}$$

It should be noted that for both types of implementations, either with or without CLA, it is still necessary to have the possibility of updating the partial remainder by adding both positive and negative terms (e.g. $+d$ and $-d$ in the case of division), a task which is assigned to the 1's complementor block COM in Fig. 1. Although other alternatives not needing COM exist when considering division, the block COM becomes necessary for square root, where the updating of the partial remainder must be performed according to the partially developed square root value, which in turn is not fixed to a given constant, but changes as the iterations evolve. The "correction" into 2's complement representation can be performed in a similar way to the one used in [15]. The block COM, can be implemented by a set of EX-OR gates, driven by an enable line, which below is referred to as f.

4.2 Proposed architecture

The reference model for our architecture is shown in Fig. 3, and is designed to implement the algorithm illustrated in section 2.2, by introducing some particular concepts and blocks which let the architecture to be particularly efficient. First of all, the partial remainder is stored in a redundant form which is different from the usual carry and sum form used in Fig. 1: register P contains the EXOR of the carry and sum bits with the same weight, which are output by the modified carry save adder MCSA, while register G stores the AND operation performed on the same bits. In other words, P holds the $p_{i,j}$ and G holds the $g_{i,j}$ bits as defined by relations (10). The contents of P and G are shifted

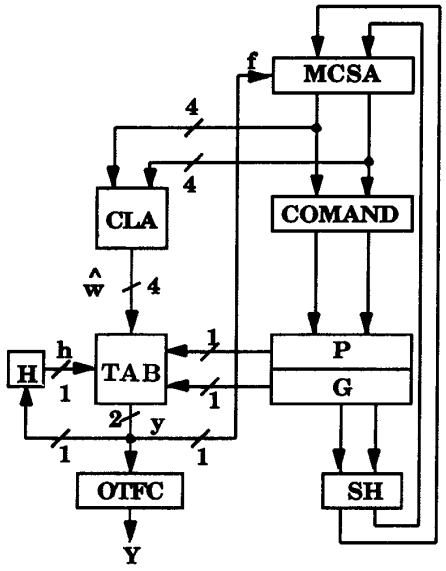

Figure 3: Architecture for fast division and square root

one position left by suitable wiring and then enter the block MCSA, i.e. a modified carry save adder. Also d (i.e. the divisor) and f (the enable signal for the 1's complementation) enter MCSA, and it is worth remembering that $f = 0$ has the implicit meaning to take d, while $f = 1$ has the meaning that the 1's complement of d is taken. The block MCSA plays an important role in our architecture. Let us consider the boolean functions for computing the "next" carry and sum of the partial remainder when a selection different from the digit 0 has been performed at the previous iteration:

$$\begin{aligned} s_{i+1,j} &= c_{i,j} \oplus s_{i,j} \oplus (f \oplus d_j) \\ c_{i+1,j} &= c_{i,j}(f \oplus d_j) + s_{i,j}(f \oplus d_j) + c_{i,j}s_{i,j} \end{aligned} \tag{11}$$

where d_j denotes the $j-th$ bit of the divisor d. Since the EXOR operator is associative, it is possible to rewrite expressions (11) as functions of $p_{i,j}$ and $g_{i,j}$ only.

$$\begin{aligned} s_{i+1,j} &= p_{i,j} \oplus f \oplus d_j \\ c_{i+1,j} &= g_{i,j} + f\bar{d}_j p_{i,j} + \bar{f}d_j p_{i,j} \end{aligned} \tag{12}$$

The block MCSA implements the functions of sum and carry given by (12), with a complexity comparable with that of the usual scheme of the carry save adder (e.g. CSA in Fig. 1) implementing relations (11) where, however, $(f \oplus d_j)$ represents a single variable (created by the block COM) and not an operation. The MCSA's output is the carry and sum representation of the new

partial remainder, and corresponds to the point where the carry lookahead adder CLA takes the most significant bits of the carry save representation of the partial remainder, just as is done in the architecture of Fig. 1. This ensures the algorithm to be exactly the same for both schemes (however, when $y_{i-1} \neq 0$), although in Figures 1 and 3 different representations of the partial remainder are retrieved in the registers. The carry and sum representation coming out from MCSA is transformed in EXOR-AND representation by the block COMAND; the outputs are stored in P and G respectively, and the unit becomes ready for a new iteration. The role of TAB is to implement the selection rules of our algorithm defined in section 2.2, while the block H is a single bit register, destined to hold the information required by the algorithm on the previous digit selection. In particular, the following convention can be adopted: during iteration i, the block H stores 0 if $y_{i-1} = 0$ and 1 if $y_{i-1} \neq 0$. The boolean variable associated with the contents of the register H is denoted with h. The design of TAB plays a key role in the whole architecture, and must be split into two cases: when a CLA is, or is not, used. We again consider the digit to be selected as being represented in sign and magnitude, with the same notation of section 4.1. When a CLA is used to perform the assimilation, from (8) we have

$$\begin{aligned} q_s &= r_{i,3}h + \bar{h}g_{i,0}^{-} \\ q_m &= (\overline{r_{i,2}r_{i,1}r_{i,0}})h + \bar{h}p_{i,0}^{-} \end{aligned} \qquad (13)$$

Conversely, when a CLA is not used, the selection functions are again given by (9) since the assimilation is carried out by the table TAB itself. and it is not necessary for h to enter the table TAB. The block OTFC is used to perform the on-the-fly conversion into a non redundant form, by following the guidelines given by Ercegovac and Lang in [3]. Observe that, at the end of all the iterations, the remainder of the division operation, can be obtained directly from the output lines of the block MCSA in carry save form. Finally, a variable duration clock should be used in the architecture, to regulate the timings of *normal* and *fast* iterations. This circuit can be easily implemented, provided that the longer cycle is an integer multiple of the shorter. In this case, the selection of a non-zero digit disables the clock for P and G registers, for a number of cycles needed to obtain the required period; this can be implemented straightway by using a counter or a shift register used as a delay line for the clock enabling signal. As a small ratio between the two clocks is expected, the number of additional flip-flop required is only a small percentage of the $2n$ memory elements used to implement P and G registers, hence it will influence only to a limited extent the cost of the whole unit. The evaluation of the whole architecture will be discussed in section 5.

4.3 Remarks and extension to square root

At this point, the operating principle of the architecture of Fig. 3 turns out to be very simple. Our architecture behaves as a "classical" unit, by performing *normal* iterations, until a digit selection $y_{i-1} = 0$ occurs. In correspondence of this event, the $i-th$ iteration is carried out faster than *normal* iterations, since it is not necessary to pass through any carry save updating. The initialization phase is the same as in the "classical" architectures, as when a new computation starts, the register P is loaded with the non redundant representation of the dividend (or radicand, for square root), while the register G is loaded with all 0's. The extension to square root is straightforward, since as we have seen in section 3 the algorithm is the same, and therefore the sharing of our unit between division and square root is implemented exactly in the same way as is done in the other architectures for both division and square root.

5 Evaluation

The complete evaluation of the proposed architecture involves two aspects: hardware requirements and performance evaluation. Concerning the hardware, only very limited additional resources are required by our architecture shown in Fig. 3 with respect to the "classical" unit in Fig. 1. In particular, they are: *i)* a "line" of AND gates to implement a part of the block COMAND, *ii)* a few flip-flops to implement the register H and the variable duration clock, *iii)* the amount of hardware necessary for the selector to permit to permit the loading through or bypass the CSA, and *iv)* (only when the CLA is used), a few logical gates so as to implement the slightly more complex selection functions of TAB with respect to the table T. The performance evaluation must take into account the two separate cases when a CLA is, or is not, used to assimilate the most significant bits of the partial remainder.

5.1 Architecture with carry lookahead adder

By looking at Fig. 3 we observe that the blocks COMAND and CLA operate in parallel. In fact, as we have seen in section 4.2, CLA is a 4 bit carry lookahead adder and COMAND is a line of EXORs in parallel to a line of AND gates, and hence it is reasonable to assume that, independently of the implementation technology, the delay of the CLA is larger than or, equal to, the delay of the block COMAND. Therefore, the execution time of a *normal* iteration of the proposed architecture in Fig. 3 is $T_{new,norm,CLA} = t_{MCSA} + t_{CLA} + t_{TAB} + t_{reg,PG}$ where, in general, t_X denotes the delay of the block X in the architecture in Fig. 3, and $t_{reg,PG}$ the delay for loading the registers P and G (including the delay of a selector to permit the loading through or bypass the CSA). On the other hand, the execution time of an iteration of the "classical" architecture is $T_{old,CLA} = t_{CSA,COM} + t_{CLA} + t_T + t_{reg,CS}$, where $t_{CSA,COM}$ is the total delay required for a carry save sum/subtraction and $t_{reg,CS}$ is the delay for loading the registers C and S. For *fast* iterations the execution time of our architecture becomes $T_{new,fast,CLA} = t_{TAB} + t_{reg,PG}$.

5.2 Architecture without carry lookahead adder

By removing from Fig. 3 the block CLA we observe that the critical path passes through MCSA, COMAND, the register load and TAB. The execution time of

a *normal* iteration of the proposed architecture becomes $T_{new,norm,noCLA} = t_{MCSA} + t_{COMAND} + t_{TAB} + t_{reg,PG}$. Let us consider now the architecture in Fig. 1 when the CLA has been removed. From the expressions of the selection functions (9) we see that they require the knowledge of the terms $p_{i,j}$ and $g_{i,j}$, as they are defined in (10). This is equivalent to considering the functions for digit selection in the architecture in Fig. 1 as being implemented by the cooperation of a block COMAND (to compute the terms of (10)), and a table T to implement the functions (9). Therefore, the execution time of an iteration of the "classical" architecture in Fig. 1 is $T_{old,noCLA} = t_{CSA,COM} + t_{COMAND} + t_T + t_{reg,CS}$. It is worth remembering that as we have seen in section 4.2, both tables TAB and T implement the same relations (9). For *fast* iterations the execution time of our architecture becomes $T_{new,fast,noCLA} = t_{TAB} + t_{reg,PG}$.

5.3 Average execution time

From the previous computations, it emerges that technological factors determine whether *normal* iterations of the proposed architectures are longer or shorter than the iteration delays of the "classical unit", although in general it is expected for this difference to be small. Anyway, even if the *normal* iterations could have a longer duration, the average performances may be improved if a sufficient number of *fast* iterations is executed. The aim of this section is to determine the average execution time, that is to say how much the *fast* iterations contribute to reducing the average iteration time. Since a pure theoretical analysis is very complicated, we have preferred to direct our efforts towards the simulation in order to detect the number of occurrences of the *fast* iterations. We have first considered the square root, by performing an exaustive type of simulation, that is to say, we have considered all the possible radicands expressed on a given number of bits. For our purposes, the generic radicand x has been taken into account as expressed in carry assimilated form. Our simulation has explicitly considered the behavior of the proposed architectures (irrespective of whether the CLA is used). This means that, in correspondence to the selection of a 0, no sum operation has been performed. The results of the simulation for different lengths of the radicands and for different assimilations of the most significant bits of the remainder, are reported in Table 2. From Table 2 we see that, as the number of bits of the radicand increases, the percentage of 0 selections is approximately 28% when an assimilation of 4 bits of the result is considered by the digit selection process. As it will be explained in section 5.4, higher percentages are obtained with the increase of the number of bits of the remainder which are assimilated. Basically, we have used the same assumptions for simulating division as those adopted for square root. The results of the simulation for different lengths of the dividends and for different assimilations of the most significant bits of the remainder, are reported in Table 3. It should be observed that the rightmost column in Table 3 (i.e. corresponding to the whole partial remainder represented in carry assimilated form), reflects approximately the bound given by Freiman in [7] on the number of digits produced per iteration by the Robertson's method. In fact, from [7], it is known that one iteration produces (on the average) 8/3 binary digits. This corresponds to having $(1 - 8/3) = 5/3$ selections of the digit zero every 8/3 binary digits produced. Therefore, with Robertson's method, according to Freiman's results, the percentage of 0-selections is 5/8=62.5%, which is approximately the same as the values shown in the CLA_N column in Table 3. From Table 3 we observe that, as the number of bits of the radicand increases, the percentages of selecting a 0 approach approximately 35% when the digit selection process considers an assimilation of 4 bits of the result. Higher percentages are obtained with the increase in the number of bits of the remainder which are assimilated.

5.4 General remarks

From the previous analysis it is clear the importance of the results of Tables 2 and 3. In fact, as we have seen, the percentages of 0-selections in Tables 2 and 3 offer a criterion for evaluating how different implementation technologies can favour the average execution time of the proposed algorithm and architecture, with respect to the delay of the "classical" architecture.

Several other conclusions can be deduced from an analysis of Tables 2 and 3, which involve the choice among a class of different architectural solutions for implementing our unit. The most important is that assimilating more bits of the partial remainder, than the strictly necessary (i.e. 4), has the effect of enlarging the region of the truncated partial remainder for which the digit 0 can be selected. This corresponds to passing from the region $[-1/2, -1/2]$ of a 4 bit assimilation, to $[-1/2, 0]$, $[-1/2, 1/4]$ and $[-1/2, 3/8]$ of 5, 6 and 7 bit assimilations, respectively. If we assume that the maximum size of these regions is being considered, then the final effect is that the percentage of selecting a 0 increases as the number of bits of the partial remainder which are assimilated increases, because of both the lower truncation error and of the larger region for a 0-selection. To take full advantage of the larger region for a 0 selection, the selection table TAB must consider all the bits of the assimilated part of the partial remainder, i.e. 5, 6 and 7 bits respectively in the case of 5, 6 and 7 bit assimilations. Although this increases the complexity of the implementation of the table TAB, it is necessary to consider all the assimilated bits of the partial remainder. In fact, as we have observed with simulation, the increase in the percentages of selecting 0's remains relatively small if the assimilated part of the partial remainder is still examined on "only" 4 bits of the "classical" architecture. For example, with a 7 bit assimilation and an inspection of 4 bits, the percentage of selecting a 0 is equal to 25.954%, that is about one half the corresponding value in Table 2, i.e. 51.724%, and does not substantially differ from the value 25.854% reported in the column of the 4 bit assimilation. This implies that the most significant role in the increase of the percentages of selecting 0's, is played by the enlargement of the regions for the 0 selection, and not by the assimilation of our extended subset of the partial remainder. For this reason, in our implementations we have considered that all the assimilated bits enter TAB.

Table 2: Square root: percentages of selection of 0

length of the radicand	length of the result	iterations performed	percentage of 0's				
			CLA_4	CLA_5	CLA_6	CLA_7	CLA_N
8 bits	5 bits	768	17.083	34.115	41.797	44.010	45.443
10 bits	6 bits	3840	20.521	36.667	44.271	46.458	48.073
12 bits	7 bits	18432	22.439	38.542	46.050	48.405	49.799
14 bits	8 bits	86016	23.865	39.983	47.363	49.805	51.267
16 bits	9 bits	393216	24.976	41.073	48.372	50.877	52.328
18 bits	10 bits	1769472	25.854	41.962	49.184	51.724	53.211
20 bits	11 bits	7864320	26.551	42.681	49.851	52.392	53.912
22 bits	12 bits	34603008	27.161	43.276	50.396	52.950	54.496
24 bits	13 bits	150994944	27.660	43.775	50.857	53.412	54.982
26 bits	14 bits	654311424	28.094	44.203	51.247	53.806	55.396

Table 3: Division: percentages of selection of 0

length of the dividend	length of the result	iterations performed	percentage of 0's				
			CLA_4	CLA_5	CLA_6	CLA_7	CLA_N
8 bits	5 bits	4096	34.766	52.539	60.059	62.500	63.965
10 bits	6 bits	40960	35.518	52.393	59.658	62.256	63.633
12 bits	7 bits	393216	35.508	52.182	59.171	61.812	63.198
14 bits	8 bits	3760016	35.434	52.053	58.869	61.466	62.914
16 bits	9 bits	33554432	35.364	51.958	58.664	61.198	62.681
18 bits	10 bits	301989888	35.296	51.891	58.526	61.011	62.501

The upper bound on the percentages of selecting a 0 is reached when the partial remainder is represented in assimilated form (see the columns CLA_N in Tables 2 and 3).

By examining the numerical values of Table 2 we observe that with an assimilation of 4 bits, percentages of 0 selections of about 28% are obtained. With the assimilation of 5 bits, the percentages increase to about 44%, while with 6 bits the increase is more limited (about 51%), then becoming almost insignificant for 7 bit assimilations (i.e. about 54%). Although it is beyond the scope of this paper to evaluate the proposed architecture in such a case since it would be highly technology dependent, it is clear that there exists a knee in the performance curve vs. the number of assimilated bits, such that the advantages of a larger assimilation are almost irrelevant compared with the price to be paid for, in terms of increased hardware complexity. In fact, a larger assimilation implies the use of a larger (and slower) CLA and of a larger (and slower) selection table TAB still considering all the bits of the assimilated part of the partial remainder [9]. Task of the designer is, therefore, to evaluate among all the most convenient implementation, given the technological constraints.

6 Differences with previous works

Several works have considered the possibility to speed up the duration of the selection process by decreasing the number of additions. Key works been provided by Metze in [8], by Freiman [7] and by Wilson in [14]. However, in all these works, partial remainders are represented in non-redundant form. In our algorithm the redundancy of the SRT-like algorithms is used to achieve two goals: firstly to allow selection rules based on limited precision comparisons (because of the carry save representation of the partial remainder), and secondly to obtain "good" probabilities of selecting a 0, and then to avoid the need of additions/subtractions for updating the partial remainder. These two goals are contradictory, and in the past the former has always been given preference with respect to the latter.

In our work we have not considered any priority between these two goals. We have not studied the derivation of selection rules for minimizing the number of non zero digits of the result, in the case of carry save representations, since it can be demonstrated that, in order to obtain a *minimal representation* in the sense used by Metze [8], for a radix 2 square root with carry save adder, the complexity of the digit selection tables is comparable to that required by radix 4 square root. Conversely, the attention has been focused on the study of the statistical properties of the "classical" radix 2 division and square root methods known in the literature, and in particular on the percentage of the 0-selections. We have then slightly modified the digit selection tables of the "classical" unit known in the literature, in order to take the full advantage of the existence of the 0-selections. Moreover, by properly tuning the portion used for the redundancy which is employed for limited

length comparisons and digit selections, and the complementary portion used for the redundancy which has the role of increasing the percentage of the 0-selections, it is possible to achieve several different architectural solutions. Eventually, unlike other algorithms using the redundant representation of the partial remainder, we use a representation different from the "classical" non-assimilated carry sum form.

From these considerations, it is clear that both our algorithm and architecture differ radically from the previous proposed in the literature.

7 Conclusions

We have presented a new algorithm for shared radix 2 division and square root. The main characteristic of the proposed scheme is the subdivision of the iteration steps for the calculation of the result, into *fast* and *normal* iterations. For *fast* iteration we mean an iteration when the addition can be avoided, while for *normal* iterations we refer to all the other iterations.

Some implementation aspects have been discussed. In particular, it has been shown that a redundant representation of the remainder different from the non assimilated carry-sum form could improve the performance characteristics so that a *normal* cycle with duration close to that of a "classical" architecture can be achieved.

The proposed design methodology can be extended to architectures where larger assimilations of the partial remainder such as the "minimal" one (i.e. the 4 most significant bits) are taken into account, with the aim of increasing the average number of *fast* iterations occurring during a computation. It can be shown that, in such a case, an increased percentage of 0's selected during a computation leads to more complex selection functions. Among the proposed solutions, i.e. pure radix 2 and radix 2 with larger assimilations, it is the task of the designer to evaluate the most convenient implementation, depending on his specific technological constraints.

Acknowledgment

We thank Prof. Miloš Ercegovac for his helpful suggestions. Thanks are also due to the referees for their valuable comments.

References

[1] D. E. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," IEEE Trans. Comput., Vol.C-17, pp.925-934, October 1968.

[2] L. Ciminiera and P. Montuschi, "Higher Radix Square Rooting," IEEE Trans. Comput., Vol.39, No.10, October 1990, pp.1220-1231.

[3] M. D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," IEEE Trans. Comput., Vol.C-36, pp.895-897, July 1987.

[4] M. D. Ercegovac and T. Lang, "Division," Internal Report CS252, Computer Science Department, UCLA, 1988.

[5] M. D. Ercegovac and T. Lang, "Radix-4 Square Root Without Initial PLA,"IEEE Trans. Comput., Vol.C-39, pp.1016-1024, August 1990.

[6] J. Fandrianto, "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square-Root," Proc. 9th IEEE Symposium on Computer Arithmetic, Santa Monica, CA, pp.68-75, September 1989.

[7] C.V. Freiman, "Statistical Analysis of Certain Binary Division Algorithms," Proc. IRE, Vol.49, pp.91-103, January 1961.

[8] G. Metze, "Minimal Square Rooting," IEEE Trans. Electron. Comput., Vol.EC-14, pp.181-185, April 1965.

[9] P Montuschi and L. Ciminiera, "Fast Radix 2 Division and Square Root," Internal Report, Politecnico di Torino, Dipartimento di Automatica e Informatica, I.R. DAI/ARC 15-90.

[10] S. Majerski, "Square-Rooting Algorithms for High-Speed Digital Circuits," IEEE Trans. Comput., Vol.C-34, pp.724-733, August 1985.

[11] J. E. Robertson, "A New Class of Digital Division Methods," IRE Trans. Electron. Comput., Vol.EC-7, pp.218-222, September 1958.

[12] J.l. Smith and A. Weinberger, "Shortcut Multiplication for Binary Digital Computers," NBS Circular 591, Sec.1, pp.13-22.

[13] P. K.-G. Tu, "On-Line Arithmetic Algorithms for Efficient Implementation," PhD Dissertation, Computer Science Department, UCLA, CSD-900029, September 1990.

[14] J.B. Wilson and R.S. Ledley, "An Algorithm for Rapid Binary Division," IRE Transactions on Electronic Computers, Vol.EC-10, pp. 662-670, December 1961.

[15] J. H. P. Zurawski and J. B. Gosling, "Design of High-Speed Digital Divider Units," IEEE Trans. Comput., Vol.C-30, pp.691-699, September 1981.