

Decimal Division Using the Newton–Raphson Method and Radix-1000 Arithmetic

Mário P. Véstias and Horácio C. Neto

1 Introduction

Computer arithmetic is predominantly performed using binary arithmetic because the hardware implementations of the operations are simpler than those for decimal computation. However, many decimal fractions cannot be represented exactly as binary fractions with a finite number of bits. The value 0.1, for example, can only be represented as an infinitely recurring binary number. If a binary approximation is used instead of the exact decimal fraction, the results will not be exact even if the arithmetic is exact. Therefore, many applications, such as financial and commercial, where the results must be exact, matching those obtained by human calculations, must be performed using decimal arithmetic. Until very recently, the adopted solution was to implement decimal operations using software algorithms based on binary arithmetic. However, these software solutions are typically three or four orders of magnitude slower than binary arithmetic implemented in hardware [4]. To speed up the execution of decimal arithmetic, a few processors, such as the IBM Power6 [1], already include dedicated hardware for decimal floating-point operations.

Decimal division is one of the fundamental operations for hardware-based decimal arithmetic. Division techniques based on digit-recurrence algorithms are the most used in (binary) hardware dividers, and have also been considered in most decimal division proposals. Nikmehr et al. [14] proposed a decimal floating-point division algorithm based on high-radix SRT division. Lang and Nannarelli [10] have also implemented a decimal division unit based on the digit-recurrence

M.P. Véstias (✉)
INESC-ID/ISEL/IPL, Lisbon, Portugal
e-mail: mvestias@deetc.isel.ipl.pt

H.C. Neto
INESC-ID/IST/UTL, Lisbon, Portugal
e-mail: hcn@inesc-id.pt

algorithm. Their work was compared to a division unit based on the Newton–Raphson (NR) iterative method concluding that the implemented digit-recurrence seemed advantageous in terms of latency compared to the NR method. Vázquez et al. [19] proposed an algorithm based on the SRT digit-recurrence method and the corresponding architecture for decimal division. Their implementation has a comparable delay to that from [10] using lower hardware complexity. Wang et al. [23] proposed an arithmetic algorithm and hardware design for decimal floating-point division using an initial piecewise linear Taylor series approximation followed by modified Newton–Raphson iterations.

Digit-recurrence algorithms have been the primary choice for decimal division mainly because of the complexity of the decimal multipliers required to implement the NR iterations. However, digit recurrence algorithms only produce one decimal digit at each iteration, while the NR method ensures quadratic convergence. Also, dividers based on the NR algorithm can take advantage of existing decimal multiplier hardware.

The performance and the area of a divider based on the NR method depend mainly on the efficiency of the decimal multipliers. Two main lines have been considered in the design of the multipliers: iterative and parallel. In the iterative approach, the multiplicand is iteratively multiplied by one digit of the multiplier to generate a partial product. Partial products are then added to produce the final decimal result. A few decimal multipliers have been proposed based on iterative units, such as [7, 8, 18]. Parallel decimal multipliers have also been recently proposed in [5, 9, 20, 21]. While parallel decimal multipliers are faster, they require much more hardware resources than iterative ones.

In this work, a new iterative decimal hardware divider is proposed. The division algorithm used is based on the calculation of the reciprocal of the divisor using the Newton–Raphson method, followed by a final multiplication by the dividend to obtain the quotient. The NR algorithm is implemented with two major optimizations, a new initial approximation and a new iterative decimal multiplier. Instead of using the Taylor series expansion, as in [23], the initial approximation of the reciprocal is calculated using piecewise minimax polynomials. This allows to reduce the number of NR iterations and therefore the size and the latency of the multipliers and consequently of the divider. All multiplications are done using very efficient iterative radix-1000 arithmetic. The partial multiplications of the radix-1000 multipliers are directly implemented using the available binary multipliers, therefore significantly improving the overall performance. Further, the use of an internal radix-1000 representation significantly simplifies the binary to/from decimal conversions.

This chapter is organized as follows. Section 2 describes the algorithm used to compute the reciprocal of a decimal number and the error analysis of the method. Section 3 describes the design of the decimal divider. Section 4 provides results with and without embedded multipliers and presents a comparison with a state-of-the-art iterative divider.

2 Reciprocal Computation

The reciprocal of the divisor, $\frac{1}{x}$, is calculated using the Newton–Raphson iterative method. The first iteration uses an initial seed, herein obtained using a piecewise linear approximation based on minimax polynomials. The method converges quadratically, that is, the error of the approximation decreases quadratically with the number of iterations.

2.1 Initial Polynomial Approximation

The minimax polynomial is the approximating polynomial which has the smallest maximum error from the given function. According to the Chebyshev alternation theorem [17], the minimax degree-1 approximation to a given function in a specific interval is the 1st order polynomial that has maximum error at the interval extremes and, with alternate sign, in one interior point.

The 1st order minimax polynomial

$$p^{(1)}(x) = b + m(x - X_0) \quad (1)$$

that approximates the function $\frac{1}{x}$ in the interval $[X_0, X_1]$ can be obtained in direct form [12], and its coefficients are

$$b = \frac{1}{2 X_0} - \frac{1}{2 X_1} + \frac{1}{\sqrt{X_0 X_1}} \quad (2)$$

$$m = -\frac{1}{X_0 X_1} \quad (3)$$

and its maximum error is

$$|E_{p1}| = \frac{1}{2 X_0} + \frac{1}{2 X_1} - \frac{1}{\sqrt{X_0 X_1}} \cdot x \quad (4)$$

2.2 Newton–Raphson Iterations

Given the divisor x , the Newton–Raphson calculates its reciprocal $\frac{1}{x}$ by finding the zero of the equation $f(y) = \frac{1}{y} - x$ using an iterative process. The NR iterations to obtain $\frac{1}{x}$ are given by

$$y^{(i+1)} = y^{(i)} (2 - y^{(i)} x), \quad (5)$$

Table 1 Maximum approximation errors for different interval sizes

| Interval size | $\max(E_{p1})$ | $\max(E_{NR_1})$ | $\max(E_{NR_2})$ |
|---------------|-----------------------|------------------------|------------------------|
| 10^{-2} | 0.11×10^{-1} | 0.13×10^{-4} | 0.19×10^{-10} |
| 10^{-3} | 0.13×10^{-3} | 0.16×10^{-8} | 0.24×10^{-18} |
| 10^{-4} | 0.13×10^{-5} | 0.16×10^{-12} | 0.25×10^{-26} |

where the initial value $y^{(0)}$ is the initial approximation to $\frac{1}{x}$, herein computed by (1). If the error $E_{NR}^{(i)}$ (theoretical error without truncation) at iteration i is given by

$$E_{NR}^{(i)} = \frac{1}{x} - y^{(i)}$$

then the error $E_{NR}^{(i)}$ at iteration $i + 1$ is given by

$$E_{NR}^{(i+1)} = x (E_{NR}^{(i)})^2 \quad (6)$$

[15].

As, for each piecewise approximation, x is a number in the interval $[X0, X1[$ then $x < X1$, and the NR iteration errors (in this case, the first two iterations) are upper-bounded by

$$E_{NR}^{(1)} < X1 (E_{p1})^2 \quad (7)$$

$$E_{NR}^{(2)} < X1^2 (E_{p1})^4. \quad (8)$$

Table 1 shows the upper bounds for the reciprocal approximation errors considering that the normalized divisor interval $[0.1, 1[$ is divided into subintervals of size 10^{-1} .

As shown, the use of subintervals of size 10^{-3} is sufficient to provide an error lower than 10^{-8} after one NR iteration, and an error lower than 10^{-18} after two NR iterations.

2.3 Truncation Errors

In practice and to reduce the size of the lookup table and of the multipliers, the arithmetic operators are implemented with less than full precision. In this case (1) becomes

$$p^{(1)}(x) = b + \varepsilon_{T0} + (m + \varepsilon_{T1})((x - X0) + \varepsilon_{T2}), \quad (9)$$

where ε_{T0} , ε_{T1} , and ε_{T2} are the truncation errors of b , m , and $(x - X0)$, respectively.

Given the input subinterval $[X0, X1[$, the maximum truncation error of the first-order polynomial approximation becomes therefore upper bounded by

Table 2 Number of fractional digits used for each operand

| 1st order polynomial | $b = 5$ | $m = 2$ | $(x - X0) = 7$ |
|----------------------|---------------|----------|-----------------|
| 1st NR iteration | $y^{(0)} = 5$ | $x = 11$ | $y^{(0)}x = 10$ |
| 2nd NR iteration | $y^{(1)} = 9$ | $x = 16$ | $y^{(1)}x = 19$ |

Table 3 Upper-bounds for maximum reciprocal errors

| Divisor digits | NR iterations | Error upper-bound |
|----------------|---------------|------------------------|
| 8 | 1 | 0.34×10^{-8} |
| 16 | 2 | 0.42×10^{-17} |

$$\varepsilon_T < \varepsilon_{T0} + (X1 - X0) \varepsilon_{T1} + \frac{\varepsilon_{T2}}{X1X0} + \varepsilon_{T1}\varepsilon_{T2}. \quad (10)$$

If the operand size is reduced in the computation of the NR iterations (5), the iteration error E_{NR} becomes

$$E_{NR}^{(i+1)} = x (E_{NR}^{(i)} + \varepsilon_{TY})^2 + y^{(i)} \varepsilon_{TP} - (y^{(i)})^2 \varepsilon_{TX}, \quad (11)$$

where ε_{TY} , ε_{TX} , and ε_{TP} are the truncation errors of the operands $y^{(i)}$, x , and of the product $y^{(i)}x$, respectively.

The iteration error can therefore be upper bounded by

$$E_{NR}^{(i+1)} \leq \max \left(X1 (E_{NR}^{(i)} + \varepsilon_{TY})^2 + \frac{1}{X0} \varepsilon_{TP}, \frac{1}{X0^2} \varepsilon_{TY} \right). \quad (12)$$

Tables 2 and 3 show the number of fractional digits used for each operand to ensure faithful rounding after the final reciprocal computation. The resulting precision for the reciprocal, according to the error upper bounds provided by (12), is also indicated. The reciprocal result will be a number in the interval $[1, 10]$ and, therefore, will have seven fractional digits, in the case of the 8-digit result, and 15 fractional digits, in the case of the 16-digit result. Faithful rounding guarantees that the computed result is one of the two floating-point neighbors of the exact result [11, 16].

3 Implementation of the Divider

Two dividers have been implemented, one divider for 8-digit operands and another for 16-digit operands. In both cases, the quotient $q = \frac{z}{x}$ is obtained by calculating the reciprocal $\frac{1}{x}$ using the method described in Sect. 2, and then multiplying it by z .

The piecewise first-order minimax polynomial approximation of the reciprocal, $y^{(0)}$, calculated according to (1), as

$$y^{(0)} = p^{(1)}(x) = b + m \times (x - X0)$$

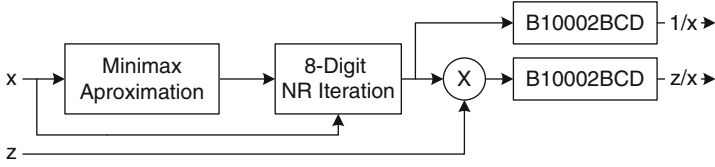


Fig. 1 Iterative 8-digit decimal divider

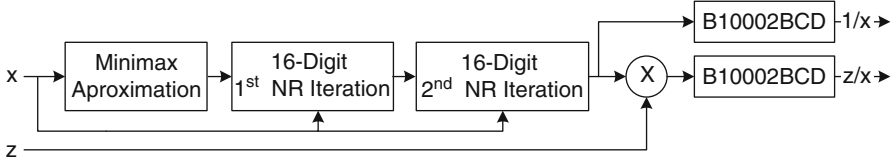


Fig. 2 Iterative 16-digit decimal divider

is the starting point of the iterative NR method for both the 8-and the 16-digit dividers. The coefficients, b and m , of the approximation polynomial are stored in a ROM and then used to calculate the initial linear approximation.

The calculation of the reciprocal requires a single iteration of the NR method, for the 8-digit division, and two iterations for the 16-digit division. This guarantees the required precision as indicated by the error upper bounds shown in Table 3.

The block diagrams for the 8-and 16-digit dividers using an initial minimax approximation and the NR method are illustrated in Figs. 1 and 2.

The outputs of both dividers are the reciprocal of the divisor and the quotient. The outputs of the NR iterations and of the last multiplier used to calculate $\frac{1}{x} \times z$ are numbers represented in radix-1000. Therefore, a final radix-1000 to decimal conversion is used to obtain the reciprocal and the final division result in *binary-coded decimal* (BCD).

A radix-1000 number, r , is represented with radix-1000 digits, $r = r_n r_{n-1} \dots r_1 r_0$, where each r_i digit is a decimal number from 0 to 999. Therefore a radix-1000 number has the following decimal value:

$$r = r_n \times 10^{3 \times n} + r_{n-1} \times 10^{3 \times (n-1)} + \dots + r_1 \times 10^{3 \times 1} + r_0.$$

This radix has some important characteristics. A radix-1000 number can be easily converted from/to a BCD number and radix-1000 arithmetic can be efficiently implemented using binary arithmetic. Also, radix-1000 (base 10^3) is close to 2^{10} , an important characteristic whenever binary from/to radix-1000 conversion has to be done, as will be explained in the following sections.

In the following sections, all blocks are described in detail.

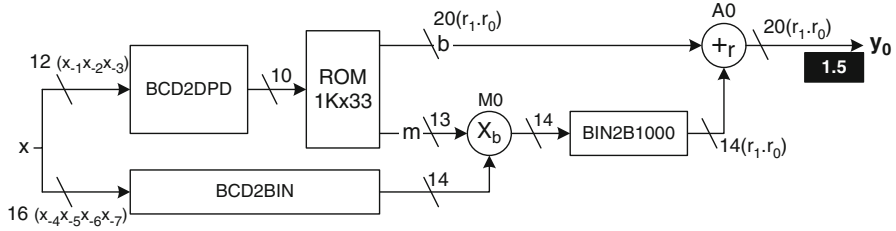


Fig. 3 Block diagram of minimax approximation unit

3.1 Minimax Approximation

To determine the initial approximation of the reciprocal, approximation intervals of size 10^{-3} have been chosen (Table 1). For each interval, the pair of coefficients (b, m) of the minimax polynomial is stored in a ROM. So, given a $x \in [0, 1[$ in the form $0.x_{-1}x_{-2}x_{-3}x_{-4}x_{-5}x_{-6}x_{-7}x_{-8}$, whose reciprocal is to be calculated, its three most significant digits, $x_{-1}x_{-2}x_{-3}$, are used to address the ROM to retrieve the coefficients b and m to be used in the polynomial calculation as follows (see Fig. 3):

$$y^{(0)} = p^{(1)}(x) = b + m \times (x - X_0),$$

where X_0 is x truncated to three fractional digits ($0.x_{-1}x_{-2}x_{-3}$).

A direct implementation of the method would require a ROM with size $2^{3 \times 4} \times k$, where k is the number of bits needed to represent the pair of coefficients (b, m) . With six digits for b and four digits for m , the ROM would have a size of $2^{12} \times 40$. To reduce the size of the ROM, the three input digits of X_0 are first converted to a 10-bit *densely packed decimal* (DPD) representation [3], to be detailed in the following section.

The operands of the multiplication in the minimax approximation are as follows

$$m \times (x - X_0) = m \times (x - 0.x_{-1}x_{-2}x_{-3}) = m \times (0.000x_{-4}x_{-5}x_{-6}x_{-7}x_{-8}).$$

One of the multiplier operands, the m coefficient, is stored in the ROM with four digits (two digits for the integer part and two digits for the fractional part). The other operand, $(x - X_0)$, is truncated to seven fractional digits, $0.000x_{-4}x_{-5}x_{-6}x_{-7}$, enough to guarantee the required precision of the initial approximation (see Table 2).

So, the product will have the format $0.0u_{-2}u_{-3}u_{-4}u_{-5}u_{-6}u_{-7}u_{-8}u_{-9}$, which is then truncated to $0.0u_{-2}u_{-3}u_{-4}u_{-5}$ (the least four significant digits are ignored).

To implement this multiplication, we ignore the fractional points and multiply the decimal digits with an integer decimal multiplier. In this case, we need a 4×4 multiplier whose result is truncated to four digits. The final four digits are the digits $u_{-2}u_{-3}u_{-4}u_{-5}$ of the minimax multiplication.

The proposed design uses a binary multiplier to implement this multiplication. Therefore, the four digits of x used in the multiplication, x_{-4} , x_{-5} , x_{-6} , x_{-7} , must be converted to binary (BCD2BIN component) while the m operand is retrieved from the ROM already in binary format.

The result of the multiplication is then added to the coefficient b , which is represented with one digit for the integer part and five digits for the fractional part. A radix-1000 adder is used, since the NR iterations use radix-1000 multipliers. Therefore, the b coefficient is stored in the ROM in radix-1000 format. However, the output from the binary multiplier is a binary number, and therefore, it must be converted to radix-1000 and truncated as stated above.

To optimize the binary to radix-1000 conversion of the output of the multiplier, followed by the truncation of the least four significant digits, the four digits of the m operand are first multiplied by a constant. The constant is determined as explained next. Given two decimal operands, $op1$ and $op2$, with four digits each, the four digits truncation of $op1 \times op2$ is given by

$$\frac{op1 \times op2}{10^4}$$

which can be rewritten as

$$op1 \times \frac{2^{13}}{10^4} \times \frac{op2}{2^{13}} = mc \times \frac{op2}{2^{13}}, \quad (13)$$

where $mc = op1 \times \frac{2^{13}}{10^4}$.

While the original coefficient m ($op1$) is representable with 14 bits, the coefficient mc needs only 13 bits. So, mc is stored instead of m . Besides, the 27-bits at the multiplier output are simply shifted 13 bits to do the four digits truncation. The remaining 14 bits from the multiplier output are then converted to radix-1000 (BIN2B1000 component) and added to b .

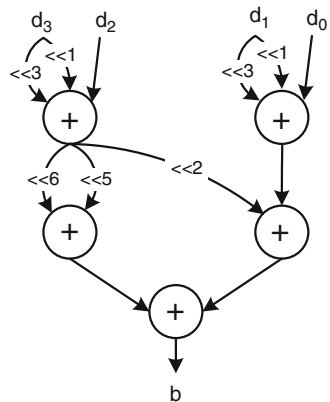
Considering these optimizations, the size of the ROM is reduced to $2^{10} \times 33$.

3.1.1 BCD to DPD Conversion

The *DPD* encoding [3] is a specific format to compress three decimal digits into 10 bits, instead of the 12 bits required using simple BCD (one digit in 4 bits). The *DPD* encoding has been developed such that the compression can be implemented using only very simple boolean operations.

The *DPD* converter transforms a group of three BCD digits ($D_2D_1D_0$, where $D_2 = abcd$, $D_1 = efgh$ and $D_0 = ijkm$) into 10-bit numbers of the form “ $pqr\ stu\ v\ wx\ y$ ” according to logic equations (14) (as proposed in [3]). All the required logic functions have five or less variables, and therefore each function can be implemented using at most a single 6-input LUT (e.g., in Virtex6 FPGA) or two 4-input LUTs and a multiplexer (e.g., in Virtex4 FPGA).

Fig. 4 Block diagram of the decimal to binary converter



$$p = b + a \cdot j + a \cdot f \cdot i$$

$$q = c + a \cdot k + a \cdot g \cdot i$$

$$r = d$$

$$s = f \cdot \bar{a} + f \cdot \bar{i} + \bar{a} \cdot e \cdot j + e \cdot i$$

$$t = g \cdot \bar{a} + g \cdot \bar{i} + \bar{a} \cdot e \cdot k + a \cdot i$$

$$u = h$$

$$v = \bar{a} \cdot \bar{e} \cdot \bar{i}$$

$$w = a + e \cdot i + j \cdot \bar{e}$$

$$x = e + a \cdot i + k \cdot \bar{a}$$

$$y = m.$$

(14)

3.1.2 BCD to Binary Converter: BCD2BIN

The conversion of a 4-digit decimal, $d = d_3d_2d_1d_0$, to binary, b , is done using binary arithmetic according to

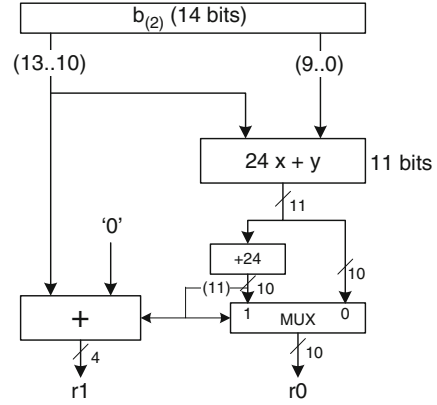
$$b = (d_3 \cdot 10 + d_2) \cdot 10^2 + d_1 \cdot 10 + d_0 \quad (15)$$

which is implemented with only adders and shifts by factoring the constants as powers of two, as

$$b = (d_3 \times 8 + d_3 \times 2 + d_2)(64 + 32 + 4) + d_1 \times 8 + d_1 \times 2 + d_0. \quad (16)$$

The final implementation requires five adders, as shown in Fig. 4 (in this figure, $\ll sh$ represents a left shift of sh bits).

Fig. 5 14-bit binary to radix-1000 converter



3.1.3 Binary to Radix-1000 Converter

The binary to radix-1000 converter (BIN2B1000 block in Fig. 3) is based on the architecture proposed in [13] to convert a 20-bit binary number to a two-digit radix-1000 number. The BIN2B1000 converter used in the minimax polynomial calculation only needs to convert a 14-bit binary number (see Fig. 5).

The circuit converts a binary number $b \in [0, 9999]$ to one decimal digit plus a digit base-1000 number r , that is

$$b = r_1 \cdot 10^3 + r_0 = r.$$

Considering that

$$b = b_1 \cdot 2^{10} + b_0$$

it follows that

$$\begin{aligned} b &= b_1 \cdot 1024 + b_0 & b_1 &\leq 10 = \frac{9999}{1024} \\ &= b_1 \cdot 1000 + \underbrace{b_1 \cdot 24 + b_0}_c & b_0 &\leq 1023, \end{aligned} \quad (17)$$

where

$$c = b_1 \cdot 24 + b_0 \quad c \leq 1215 \leftarrow 11 \text{ bits}. \quad (18)$$

From Eqs. (17) and (18), b is given by

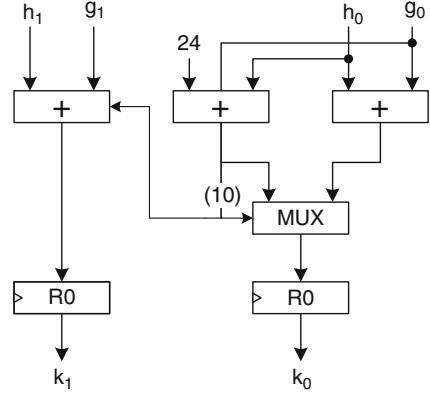
$$b = (b_1) \cdot 1000 + c \quad (19)$$

and a first approximation for the two digits is:

$$\hat{r}_1 = b_1 \leq 9 \quad \leftarrow 4 \text{ bits} \quad (20)$$

$$\hat{r}_0 = c \leq 1215 \quad \leftarrow 11 \text{ bits}. \quad (21)$$

Fig. 6 Adder radix-1000 for numbers with two radix-1000 digits



From these, the final step is to test if \hat{r}_0 is higher than 999, that is, if $\hat{r}_0 + 24$ is higher than 1023. If yes, \hat{r}_1 must be incremented by one and \hat{r}_0 must be increased by 24 to adjust the result. Otherwise, the result is already correct.

3.1.4 Adder Radix-1000

A radix-1000 adder sums two radix-1000 numbers, $g = g_2g_1g_0$ and $h = h_2h_1h_0$. The result, $k = k_2k_1k_0 = g + h$, is calculated as follows:

$$\begin{aligned}
 k'_0 &= g_0 + h_0 \\
 k'_1 &= g_1 + h_1 + Cy_0 \\
 k'_2 &= g_2 + h_2 + Cy_1 \\
 k_m &= k'_m, \quad Cy_m = 0, \quad \text{if } k'_m < 1000 \\
 &= k'_m + 24, \quad Cy_m = 1, \quad \text{otherwise.}
 \end{aligned} \tag{22}$$

The radix-1000 adder in the circuit for minimax polynomial calculation adds two operands with two radix-1000 digits each. In this particular case, the result can also be represented with only two radix-1000 digits. Therefore, the more general case given in Eq. (22) simplifies into Eq. (23).

$$\begin{aligned}
 k'_0 &= g_0 + h_0 \\
 k'_1 &= g_1 + h_1 + Cy_0 \\
 k_0 &= k'_0, \quad Cy_0 = 0, \quad \text{if } k'_0 < 1000 \\
 &= k'_0 + 24, \quad Cy_0 = 1, \quad \text{otherwise} \\
 k_1 &= k'_1.
 \end{aligned} \tag{23}$$

A block diagram of the implementation is sketched in Fig. 6. As shown, to verify if the addition of the lowest digits, k'_0 , is lower than 1000, k'_0 is added with 24 and the most significant bit of the result is checked. If it is '0' it means that result is lower than 1024 and so k'_0 is lower than 1000. This avoids the utilization of a comparator.

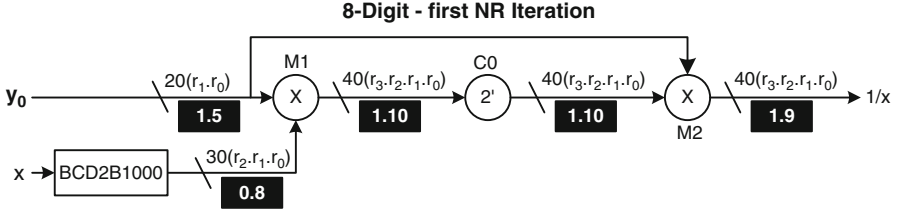


Fig. 7 NR iteration unit for the 8-digit reciprocal

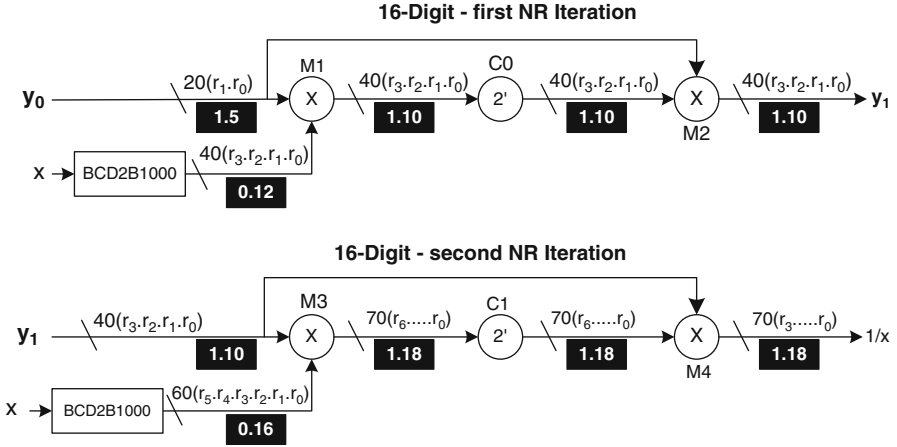


Fig. 8 First and second NR iteration units for the 16-digit reciprocal

3.2 NR Iterations

The 8-digit reciprocal is calculated with a single NR iteration:

$$\frac{1}{x} = y^{(0)} \times (2 - x \times y^{(0)})$$

while the 16-digit reciprocal calculation requires two NR iterations:

$$y^{(1)} = y^{(0)} \times (2 - x \times y^{(0)})$$

$$\frac{1}{x} = y^{(1)} \times (2 - x \times y^{(1)}).$$

Each NR iteration needs two multipliers and a circuit to calculate $(2 - w)$. However, the precision (number of digits) considered at each intermediate calculation is different for each case (see Figs. 7 and 8).

The inputs of the NR iteration unit, for the 8-digit reciprocal, are the output of the minimax approximation unit and the eight fractional digits of x in radix-1000. The block *BCD2B1000* converts the decimal number, x , to three radix-1000 digits.

The first multiplication, M1, must keep ten fractional digits plus an integer one (11 digits), and therefore four radix-1000 digits (4×3 decimal digits) are kept at the output of the multiplier. Next, the block 2' implements $2 - x \times y^{(0)}$ and maintains the same precision. Finally, the operands of the last multiplier, M2, are two radix-1000 numbers with two and four radix-1000 digits, respectively, and four radix-1000 digits are kept at the output to guarantee a result with one integer and nine fractional digits.

For the 16-digit reciprocal case, the number of fractional digits of x used in the first multiplication increases from eight to twelve, and the final result of the first iteration unit must keep ten fractional digits instead of nine. The output of the second iteration unit keeps one integer digit and 18 fractional digits.

In all implementations, the multiplications are done with iterative radix-1000 multipliers. The dimensions of the multipliers in terms of radix-1000 operands are as follows:

| Divider | M1 | M2 | M3 | M4 |
|----------|--------------|--------------|--------------|--------------|
| 8-Digit | 2×3 | 2×4 | – | – |
| 16-Digit | 2×4 | 2×4 | 4×6 | 4×7 |

The following sections describe the implementation of the iterative radix-1000 multipliers, the decimal to radix-1000 converter, and the $(2 - w)$ block.

Radix-1000 Multiplier

The radix-1000 multiplications of the divider are implemented with iterative multipliers. As referred, using parallel multipliers would turn the solution very expensive in terms of resources.

In each iteration of the multiplication, two radix-1000 digits are multiplied and the result is accumulated. This allows the utilization of a binary multiplier since a radix-1000 digit is in binary format. The results published in [22] suggest that this method is very efficient as long as binary to decimal conversions of large numbers are avoided since the size of binary to decimal converters increases more than linearly [21] with the size of the operands.

Formally, given two radix-1000 numbers, g and h , in the form

$$g = g_{n-1}g_{n-2} \dots g_2g_1g_0$$

$$h = h_{n-1}h_{n-2} \dots h_2h_1h_0$$

or

$$g = \sum_{i=0}^{n-1} g_i 10^{3 \times i}, \quad h = \sum_{i=0}^{n-1} h_i 10^{3 \times i}$$

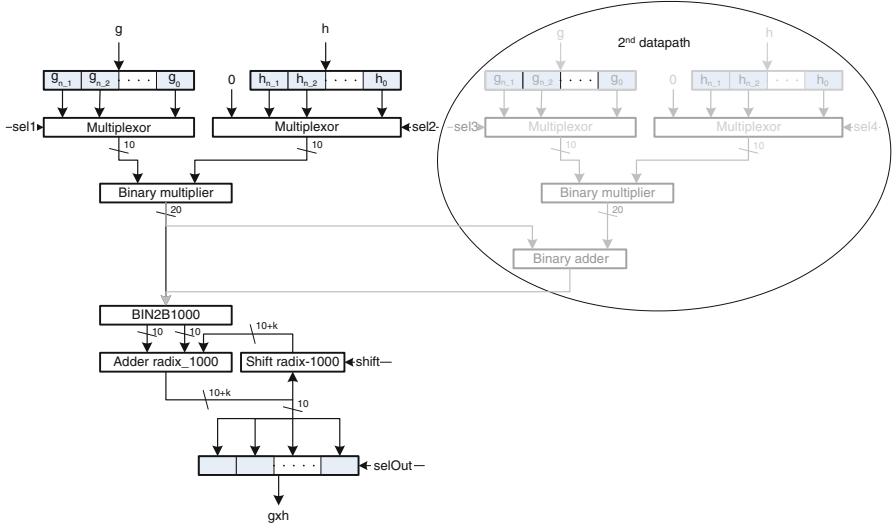


Fig. 9 Iterative radix-1000 multiplier using binary arithmetic

the iterative multiplication is given by

$$g \times h = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} g_i \times h_j 10^{3 \times (i+j)}. \quad (24)$$

The architecture of the iterative multiplier is illustrated in Fig. 9.

At each step, a pair of radix-1000 digits is selected at the input multiplexers and multiplied. The result of the multiplication is a 20-bit binary number that is converted to two radix-1000 digits, which are then accumulated also in radix-1000 with the previous result. Working with radix-1000 reduces the division by 1000 to a shift of a digit radix-1000.

The shifts take place according to the multiplication algorithm. According to Eq. (24), the first shift happens after one accumulation ($g_0 h_0$), the second shift after two accumulations ($g_0 h_1 + g_1 h_0$), until a maximum of n accumulations, followed by $n - 1$, $n - 2$ accumulations, and so on.

To reduce the number of iterations, a second internal path for the parallel calculation of partial products may be used. This second path is only used in the second NR iteration, as explained later.

The following sections describe the implementation of the binary to radix-1000 converter ($BIN2B1000$ component) and the radix-1000 adder with accumulation.

Binary to Radix-1000 Converter

In the single path multiplier, the input of the binary to radix-1000 converter is 20 bits large, and in the double path multiplier is 21 bits large.

The input of the 20 bit converter is a binary number $b \in [0, 999999]$ and the output is a two radix-1000 digits number, r_1, r_0 , that is

$$b = r_1 \cdot 10^3 + r_0 = r.$$

Considering that

$$b = b_1 \cdot 2^{10} + b_0$$

it follows that

$$\begin{aligned} b &= b_1 \cdot 1024 + b_0 & b_1 &\leq 974 = \frac{999 \times 999}{1024} \\ &= b_1 \cdot 1000 + \underbrace{b_1 \cdot 24 + b_0}_c & b_0 &\leq 1023, \end{aligned} \quad (25)$$

where

$$\begin{aligned} c &= b_1 \cdot 24 + b_0 & c &\leq 24399 && \leftarrow 15 \text{ bits} \\ c &= c_1 \cdot 1024 + c_0 \\ &= c_1 \cdot 1000 & c_1 &\leq 23 && \leftarrow 5 \text{ bits} \\ &\quad + c_1 \cdot 24 & c_1 \times 24 &\leq 23 \times 24 \\ &\quad + c_0 & c_0 &\leq 1023 && \leftarrow 10 \text{ bits.} \end{aligned} \quad (26)$$

From equations (25) and (26), b is given by:

$$b = (b_1 + c_1) \cdot 1000 + c_1 \cdot 24 + c_0 \quad (27)$$

and a first approximation for the two digits is

$$\hat{r}_1 = b_1 + c_1 \leq 997 \quad \leftarrow 10 \text{ bits} \quad (28)$$

$$\hat{r}_0 = c_1 \cdot 24 + c_0 \leq 1585 \quad \leftarrow 11 \text{ bits,} \quad (29)$$

where

$$\begin{aligned} \hat{r}_1 &\in [r_1 - 1, r_1] \\ \hat{r}_0 &\in [0, 1575]. \end{aligned}$$

From these, the final step is to test if \hat{r}_0 is higher than 999, that is, if $\hat{r}_0 + 24$ is higher than 1023. If yes, \hat{r}_1 must be incremented by one and \hat{r}_0 must be increased by 24 to adjust the result. Otherwise, the result is already correct.

The converter consists of a set of adders and a ROM to convert a $k \times 2^{10}$ number to radix-1000, as shown in Fig. 10.

The input of the 21-bit converter is a binary number $b \in [0, 2 \times 999 \times 999]$, and the outputs are a two radix-1000 digits number, $r_1 r_0$, and a carry out (Cout). The converter is similar to the case with 20 bits, except that an extra step is needed to extract the carry out from the second digit (see Fig. 11).

Fig. 10 20-bit binary to radix-1000 converter

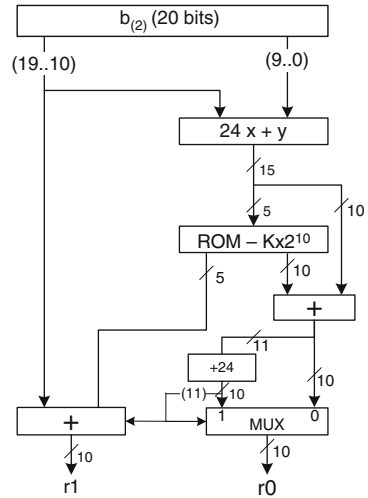
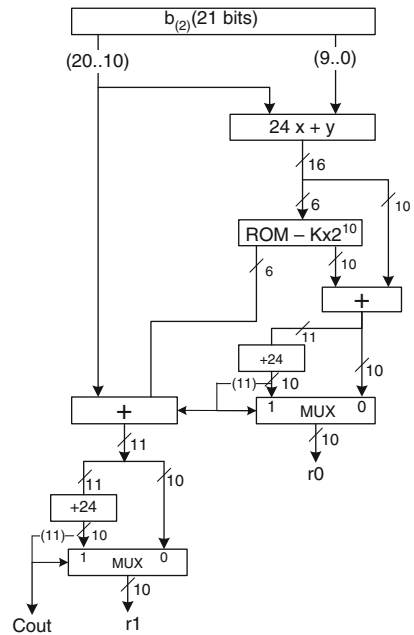


Fig. 11 21-bit binary to radix-1000 converter



3.2.1 Radix-1000 Accumulator

The radix-1000 accumulator adds radix-1000 digits in the interval $[0, 999 \times 999]$ for the single datapath and in $[0, 2 \times 999 \times 999]$ for the double datapath multiplier. Considering an $m \times n$ radix-1000 multiplier, $\max(n, m)$ accumulations can occur before a shift takes place. The biggest multiplier used to implement the divider is

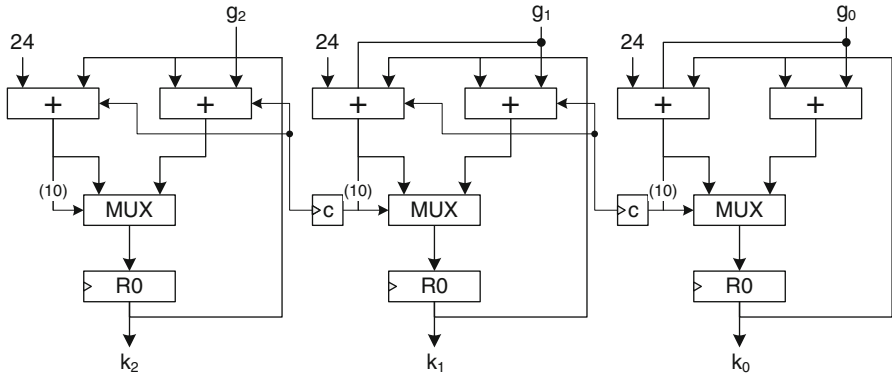


Fig. 12 Radix-1000 accumulator

a 6×7 multiplier. Therefore, a three radix-1000 digits accumulator is enough to implement any of the iterative multipliers utilized to implement the divider.

Given two radix-1000 numbers, $g = g_2g_1g_0$ and $h = h_2h_1h_0$ the addition $k = k_2k_1k_0 = g + h$, is calculated as follows:

$$\begin{aligned}
 k'_0 &= g_0 + h_0 \\
 k'_1 &= g_1 + h_1 + Cy_0 \\
 k'_2 &= g_2 + h_2 + Cy_1 \\
 k_m &= k'_m, \quad Cy_m = 0, \quad \text{if } k'_m < 1000 \\
 &= k'_m + 24, \quad Cy_m = 1, \quad \text{otherwise.}
 \end{aligned} \tag{30}$$

According to the block diagram of the iterative decimal multiplier (Fig. 9), the radix-1000 number obtained after conversion of the result from the binary multiplier is added to the previous accumulation. A direct implementation of the algorithm would have a high carry propagation delay. To improve the performance, we have implemented a carry save adder and used a similar scheme to test if the number is higher than 999.

The final implementation is shown in Fig. 12. Each input digit is added to both the previously accumulated digit of the same arithmetic weight and with 24. The results from both adders are multiplexed according to the most significant bit of the plus 24 adder. The result and the carry out are registered to implement carry save addition. In the case of a single datapath multiplier, $g_2 = "0"$. For a double datapath, $g_2 = Cout$, where $Cout$ comes from the binary to radix-1000 converter.

3.2.2 Decimal to Radix-1000 Converter

Decimal to radix-1000 conversion is straightforward. Each radix-1000 results from the conversion of three decimal digits to binary. Formally, given a decimal number

$d_{n-1}d_{n-2}\dots d_1d_0$, the radix-1000 equivalent, $r_{n-1}r_{n-2}\dots r_1r_0$, is determined as follows:

$$r_i = DEC2BIN(d_{2+3\times i}d_{1+3\times i}d_{0+3\times i}), \quad i = 0, 1, 2, \dots \quad (31)$$

The decimal to binary conversion of three digits is obtained, using binary arithmetic, as

$$b = d_210^2 + d_110 + d_0 \quad (32)$$

whose implementation is straightforward.

3.2.3 Two's Complement of a Radix-1000 Number

Each iteration of the NR has a multiplicative factor of the form $(2 - w)$, where $w = w_{n-1}w_{n-2}\dots w_1w_0$ is the output of the first multiplier of the iterative unit represented in radix-1000. w has one integer digit equal to 0 or 1. Given a number in radix-1000 with n digits, w , and knowing that the most significant digit is only 0 or 1, $(2 - w) = c = c_{n-1}c_{n-2}\dots c_1c_0$ is calculated as follows:

$$\begin{aligned} c'_0 &= 1000 - w_0 \\ c_0 &= 0, \quad Cy_0 = 1, \quad \text{if } c'_0 = 1000 \\ &= c'_0, \quad Cy_0 = 0, \quad \text{otherwise} \\ c'_1 &= 999 - w_1 + Cy_0 \\ c_1 &= 0, \quad Cy_1 = 1, \quad \text{if } c'_1 = 1000 \\ &= c'_1, \quad Cy_1 = 0, \quad \text{otherwise} \\ &\dots = \dots \\ c_{n-1} &= 1 - w_{n-1} + Cy_{n-2}. \end{aligned} \quad (33)$$

3.3 Binary to BCD Converter

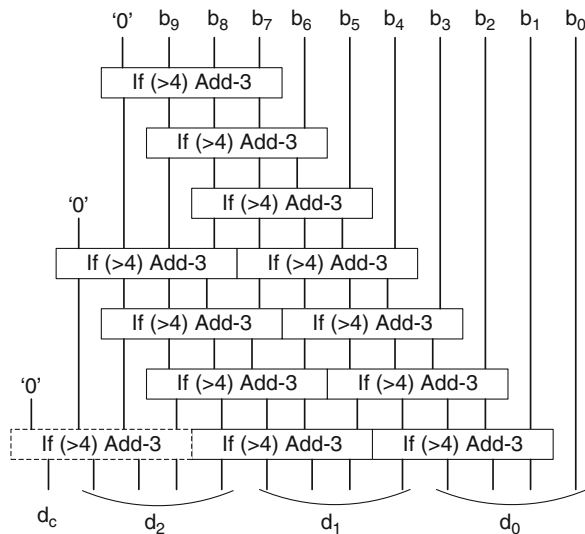
The BCD converter transforms a 10-bit binary number, b , into a BCD number with three digits, d_2 , d_1 , and d_0 . Binary to decimal conversion is fundamentally the calculation of the following polynomial, using decimal arithmetic:

$$b = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_0 \cdot 2^0 \quad (34)$$

or

$$b = (((b_{n-1} \cdot 2) + b_{n-2}) \cdot 2 + b_{n-3}) \cdot 2 + \dots + b_0. \quad (35)$$

Fig. 13 Add-3 and shift algorithm for a 10-bit binary number



Multiplication by two is achieved with a shift towards the most significant bit. However, since the operations are in decimal whenever a bit shifts across a boundary of a digit, the digit must be corrected by adding three before the shift takes place [2] (or six after the shift). The three-digit binary to decimal converter (see Fig. 13) was implemented using this algorithm, which is usually known as the add-3 and shift algorithm.

In the converter used in the divider, the binary number to be converted is always less than 1000. Whenever this is the case, the left bottom block of the converter (dashed block in Fig. 13) can be removed since d_c is zero.

3.4 Improving the Performance of the Divider

The performance of the decimal divider depends mainly on the efficiency of the iterative multipliers. The maximum operating frequency of the multiplier is constrained by the initial datapath (multiplexer, binary multiplier and binary to radix-1000 converter) that calculates the partial products, which are successively accumulated. To improve its performance, and consequently the performance of the divider, the multiplier datapath is pipelined such that each stage is optimally balanced with the internal loop longest path. Consequently, the maximum frequency becomes constrained by the internal loop consisting of the radix-1000 accumulator.

Also, a multiplier in the datapath of the complete divider can start calculating as soon as the first partial product of the previous operation is available. Therefore, the execution of the multipliers can overlap in time. A more detailed accounting of the number of cycles is provided in the following section.

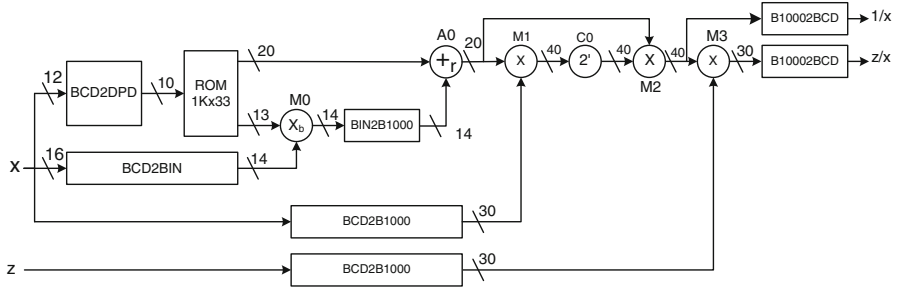


Fig. 14 Final architecture of the 8-digit divider

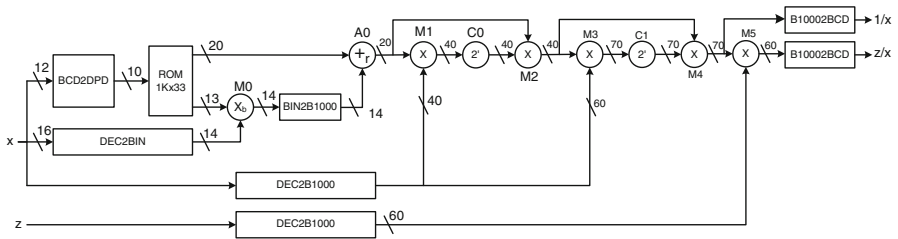


Fig. 15 Final architecture of the 16-digit divider

The complete divider architecture comprises the minimax approximation block, the NR iterations components, the final multiplier ($\frac{1}{x} \times z$), and the radix-1000 to decimal converters, as shown in Figs. 14 and 15.

The dividers provide as output results both the reciprocal and the quotient.

4 Results

The architectures of the dividers for operands of size 8 and 16 were specified in VHDL. The circuits were synthesized, placed, and routed with Xilinx ISE13.1 and implemented in a Virtex-4 SX35-12 FPGA and in a Virtex-6 VLX75T. The results were compared with an SRT-like divider from [6], which proposed a nonrestoring algorithm (alg1) and an SRT-like algorithm (alg2), both implemented in a Virtex-4 FPGA.

Two different architectures were considered for the 8-digit divider, A8Single and A8Double, with the following characteristics:

- A8Single—All multipliers use a single datapath
- A8Double—The final multiplier uses a double datapath

Three different architectures were considered for the 16-digit divider, A16Single, A16Double and A16OneDouble, with the following characteristics:

Table 4 Results for the 8-digit divider in a Virtex-4 (times in ns)

| A/B | FF | LUT | BRAMs | DSPs | Cycles | T_{clk} | Exec (cycles $\times T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|----------|------|------|-------|------|--------|-----------|------------------------------------|-------|------------------------|
| A8Single | 1371 | 2016 | 2 | 0 | 51 | 3.4 | 173 | 14 | 21 |
| A8Single | 1101 | 1605 | 2 | 5 | 51 | 3.4 | 173 | 14 | 21 |
| A8Double | 1489 | 2224 | 2 | 0 | 47 | 3.4 | 160 | 12 | 24.5 |
| A8Double | 1267 | 1783 | 2 | 5 | 47 | 3.4 | 160 | 12 | 24.5 |
| alg1[6] | 151 | 2008 | 0 | 0 | 10 | 20.5 | 205 | 10 | 4.9 |
| alg2[6] | 231 | 2612 | 0 | 0 | 10 | 16.4 | 164 | 10 | 6.1 |

Table 5 Results for the 16-digit divider in a Virtex-4 (times in ns)

| A/B | FF | LUT | BRAMs | DSPs | Cycles | T_{clk} | Exec (cycles $\times T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|-----------|------|------|-------|------|--------|-----------|------------------------------------|-------|------------------------|
| A16Single | 2098 | 2756 | 2 | 0 | 118 | 3.4 | 401 | 43 | 6.8 |
| A16Single | 1820 | 2091 | 2 | 7 | 118 | 3.4 | 401 | 43 | 6.8 |
| A16Double | 2361 | 3768 | 2 | 0 | 96 | 3.4 | 326 | 25 | 11.7 |
| A16Double | 2172 | 2718 | 2 | 10 | 96 | 3.4 | 326 | 25 | 11.7 |
| alg1[6] | 246 | 2974 | 0 | 0 | 18 | 21 | 386 | 18 | 2.6 |
| alg2[6] | 342 | 3799 | 0 | 0 | 18 | 16.6 | 300 | 18 | 3.3 |

Table 6 Results for the 8-digit divider in a Virtex-6 (times in ns)

| A/B | FF | LUT | BRAMs | DSPs | Cycles | T_{clk} | Exec (cycles $\times T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|----------|------|------|-------|------|--------|-----------|------------------------------------|-------|------------------------|
| A8Single | 1355 | 1549 | 1 | 0 | 51 | 2.6 | 135 | 14 | 27 |
| A8Single | 1009 | 987 | 1 | 4 | 51 | 2.6 | 135 | 14 | 27 |
| A8Double | 1427 | 1737 | 1 | 0 | 47 | 2.6 | 122 | 12 | 32 |
| A8Double | 1182 | 1166 | 1 | 4 | 47 | 2.6 | 122 | 12 | 32 |

- A16Single—All multipliers use a single datapath
- A16Double—Multipliers in the second NR iteration and the final multiplier use a double datapath
- A16OneDouble—Only the final multiplier uses a double datapath

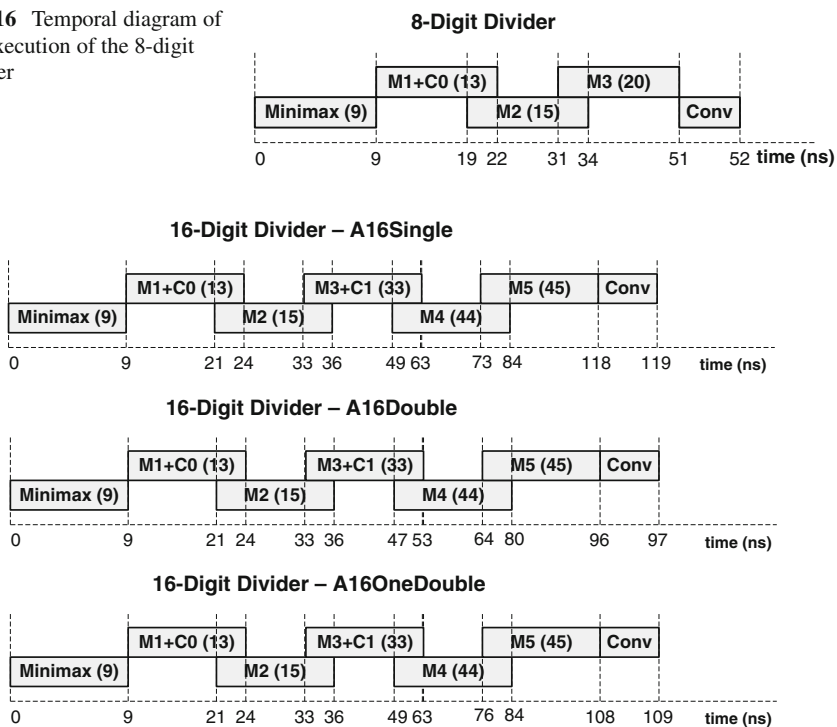
Also, for each architecture, implementations with different performance/area tradeoffs were considered, including implementations utilizing dedicated DSPs of the FPGA. The results obtained (after place and route) are summarized in Tables 4 and 5, for the Virtex-4, and in Tables 6 and 7, for the Virtex-6 FPGA.

The throughput indicated in the tables is based on the number of cycles after which a new division can start (# CBD).

The total number of cycles depends not only on the number of cycles of each block but also on the overlap of execution times of each multiplier, as shown in Figs. 16 and 17. The figures indicate the execution and the starting times (in clock cycles) of each hardware block in the datapath. Since the execution of the last multiplier overlaps only with the previous block, it is important to speed up its execution. This is the reason why the A16OneDouble architecture, where only the

Table 7 Results for the 16-digit divider in a Virtex-6 (times in ns)

| A/B | FF | LUT | BRAMs | DSPs | Cycles | T_{clk} | Exec (cycles $\times T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|--------------|------|------|-------|------|--------|-----------|------------------------------------|-------|------------------------|
| A16Single | 2545 | 2517 | 1 | 0 | 118 | 2.7 | 321 | 43 | 8.6 |
| A16Single | 2637 | 2822 | 1 | 0 | 118 | 2.6 | 309 | 43 | 8.9 |
| A16Single | 2213 | 1771 | 1 | 6 | 118 | 2.7 | 321 | 43 | 8.6 |
| A16Double | 2875 | 3156 | 1 | 0 | 96 | 2.7 | 262 | 25 | 14,8 |
| A16Double | 2934 | 3452 | 1 | 0 | 96 | 2.6 | 252 | 25 | 15,4 |
| A16Double | 2390 | 2026 | 1 | 9 | 96 | 2.7 | 262 | 25 | 14,8 |
| A16OneDouble | 2801 | 3020 | 1 | 0 | 108 | 2.7 | 294 | 30 | 12.3 |
| A16OneDouble | 2603 | 2720 | 1 | 0 | 108 | 2.6 | 283 | 30 | 12.8 |
| A16OneDouble | 2376 | 2235 | 1 | 7 | 108 | 2.7 | 294 | 30 | 12.3 |

Fig. 16 Temporal diagram of the execution of the 8-digit divider**Fig. 17** Temporal diagram of the execution of the 16-digit divider for the three different architectures

last multiplier uses a double datapath, achieves almost 50 % (10 ns) of the total performance improvement (22 ns) achieved with the architecture A16Double, where both multipliers of the second NR iteration and the last multiplier have a double datapath.

From the tables, we observe and conclude the following:

- The proposed 8×8 decimal divider without DSPs utilizes about the same area of Alg1 with a 16 % reduction in execution time. The solution using a double datapath in the final multiplier achieves an 8 % improvement in the execution time, with a 10 % increase in area.
- The solutions with DSPs have a 10 % better execution time than the implementations without DSPs and save around 20 % of LUTs.
- The throughput of the proposed 8×8 decimal divider is 4 to 5 times better compared to those obtained with alg1 and alg2.
- For the 16×16 divider, the area and execution time of both solutions are similar. In terms of throughput, the proposed divider is almost four times better.
- The execution time in Alg1 and Alg2 increases linearly with the number of digits. However, it increases more than linearly in the proposed divider. On the other hand, the area increases linearly in Alg1 and more than linearly in our proposal and in Alg2. So, for larger operands, the efficiency of the NR method will decrease compared to Alg1 and Alg2.
- Similar conclusions can be taken for the Virtex-6 implementations. These utilize less LUTs since we are dealing with 6-input LUTs instead of 4-input LUTs of Virtex-4 FPGAs. Also, the execution time is from 25 % to 30 % better compared to the implementation using a Virtex-4 FPGA.

5 Conclusions

Iterative dividers of sizes 8×8 and 16×16 based on the Newton–Raphson method with an initial minimax approximation were proposed and implemented in reconfigurable hardware.

The NR-based method for division has the advantage of producing the reciprocal that can be used for successive divisions by the same number. However, the remainder is not available.

The results show that the division based on the NR iterative method is competitive with SRT-based solutions, achieving significantly higher throughputs, as long as it uses a good initial approximation and efficient multipliers. Our approach, using the minimax polynomial for initial approximation, needs one less iteration than the Taylor-based approximation to achieve the precision required.

Also, the proposed decimal multipliers use binary multipliers. This is specially attractive when the target technology includes embedded binary multipliers, such as FPGAs.

Acknowledgements This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

References

1. POWER6 Decimal Floating Point (DFP) (2009). URL <http://www.ibm.com/developerworks/wikis/display/WikiPtype/Decimal+Floating+Point>
2. Alfke P, New B (1997) Serial code conversion between BCD and binary. In: Xilinx application note XAPP 029
3. Cowlishaw M (2002) Densely packed decimal encoding. *IEE Proc Comput Digit Tech* 149(3):102–104. DOI 10.1049/ip-cdt:20020407
4. Cowlishaw MF (2003) Decimal floating-point: algorism for computers. In: *Proceedings IEEE 6th IEEE international symposium on computer arithmetic*, pp 104–111
5. Dadda L, Nannarelli A (2008) A variant of a radix-10 combinational multiplier. In: *Proceedings IEEE international symposium on circuits and systems (ISCAS)*, pp 3370–3373
6. Deschamps JP, Sutter G (2010) Decimal division: Algorithms and FPGA implementations. In: *Proceedings IEEE southern conference on programmable logic*, pp 67–72
7. Erle MA, Schulte MJ (2003) Decimal multiplication via carry-save addition. In: *Proceedings IEEE 14th IEEE international conference on application specific systems*, pp 348–358
8. Kenney RD, Schulte MJ, Erle MA (2004) High-frequency decimal multiplier. In: *Proceedings IEEE international conference on computer design: VLSI in computers and processors*, pp 26–29
9. Lang T, Nannarelli A (2006) A radix-10 combinational multiplier. In: *Proceedings IEEE 40th international asilomar conference on signals, systems, and computers*, pp 313–317
10. Lang T, Nannarelli A (2007) A radix-10 digit-recurrence division unit: algorithm and architecture. *IEEE Transactions on Computers*, 56(6):1–13
11. Louvet N, Muller JM, Panhaleux A (2010) Newton–Raphson algorithms for floating-point division using an FMA. In: *Proceedings IEEE 20th international conference on application-specific systems architectures and processors*, pp 200–207
12. Muller JM (2006) *Elementary functions—algorithms and implementation*. Birkhauser, Basel
13. Neto HC, Véstias MP (2008) Decimal multiplier on fpga using embedded binary multipliers. In: *Proceedings IEEE 20th international conference on field programmable logic and applications*, pp 197–202
14. Nikmehr H, Phillips B, Lim CC (2006) Fast decimal floating-point division. *IEEE Trans VLSI Syst* 14(9):951–961
15. Parhami B (2000) *Computer arithmetic—algorithms and hardware designs*. Oxford University Press, Oxford
16. Rump SM, Ogita T, Oishi S (2008) Accurate floating-point summation part i: faithful rounding. *SIAM J Sci Comput* 31:189–224. DOI <http://dx.doi.org/10.1137/050645671>. URL <http://dx.doi.org/10.1137/050645671>
17. Suetin PK (2001) Chebyshev polynomials, *Encyclopedia of mathematics* edition. Springer, Berlin
18. Sutter G, Todorovich E, Bioul G, Vazquez M, Deschamps JP (2009) FPGA implementations of BCD multipliers. In: *Proceedings IEEE international conference on reconfigurable computing and FPGAs*, pp 36–41
19. Vázquez A, Antelo E, Montuschi O (2007) A radix-10 SRT divider based on alternative BCD codings. In: *Proceedings IEEE international conference on computer design*, pp 280–287
20. Vázquez A, Antelo E, Montushi P (2007) A new family of high-performance parallel decimal multipliers. In: *Proceedings IEEE 18th symposium on computer arithmetic*, pp 195–204
21. Véstias MP, Neto HC (2010) Parallel decimal multipliers using binary multipliers. In: *Proceedings IEEE southern conference on programmable logic*, pp 73–78
22. Véstias MP, Neto HC (2011) Iterative decimal multiplication using binary arithmetic. In: *Proceedings IEEE southern conference on programmable logic*, pp 257–262
23. Wang LK, Schulte M (2004) Decimal floating-point division using newton-raphson iteration. In: *Proceedings IEEE international conference on application-specific systems, Architectures and processors*, pp 84–95

<http://www.springer.com/978-1-4614-1361-5>

Embedded Systems Design with FPGAs

Athanas, P.; Pnevmatikatos, D.; Sklavos, N. (Eds.)

2013, X, 278 p., Hardcover

ISBN: 978-1-4614-1361-5