Performance/Power Space Exploration for Binary64 Division Units

Alberto Nannarelli, *Senior Member, IEEE*

Abstract—The digit-recurrence division algorithm is used in several high-performance processors because it provides good tradeoffs in terms of latency, area and power dissipation. In this work we develop a minimally redundant radix-8 divider for binary64 (double-precision) aiming at obtaining better energy efficiency in the performance-per-watt space. The results show that the radix-8 divider, when compared to radix-4 and radix-16 units, requires less energy to complete a division for high clock rates.

Index Terms—Floating-point, division, energy efficiency, digit-recurrence

1 Introduction

Division is implemented in hardware in most of contemporary processors. Two approaches are used: division by digit-recurrence and multiplicative algorithms (e.g., Newton-Raphson) [1]. The former approach is used in Intel Pentium and Core2 CPUs [2], in ARM processors [3], and in IBM FPUs [4]. The multiplicative algorithms require a multiplier plus some tables and are used in AMD processors [5], NVIDIA GPUs [6], and in Intel Itanium CPUs [7].

A comparison in terms of energy efficiency of the two approaches is done in [8] by comparing a radix-16 digit-recurrence unit (similar to the one of the Intel Core2) and a Fused Multiply-Add (FMA) floating-point unit (FP-unit) augmented with tables and by-passes to implement the Newton-Raphson division by reciprocal multiplication. The results show that the digit-recurrence approach is much more energy efficient.

Today, the greatest challenge is the so called *"power wall"*: nanometric size transistors are packed in billions in the chip, but the power requirements are such that the chip temperature can rise to a level to destroy the chip itself. Therefore, power dissipation must be limited and energy efficient units (or cores) must be designed.

Moreover, beside the temperature rise, it might become a problem to deliver enough power to all the many cores of contemporary processors. That is, not all the cores can be power supplied simultaneously, also referred as *"dark silicon"* [9].

The result is that while in the XX century area on the silicon die was a major design constraint, now that we cannot power all the die at maximum performance simultaneously, area becomes less important as part of the chip will stay *"dark"* or *"dimmed"* for some time. Therefore, power efficiency has become the main constraint in the design of high performance computing systems. In this context, two metrics characterize the energy (power) efficiency:

- the energy to complete an operation, i.e. specifically, the energy-per-division, defined as
$$E_{div} = P_{ave} \times \text{latency } [J],$$
- the performance–per–watt (PpW), defined as
$$\text{PpW} = \frac{\frac{1}{\text{latency}}}{P_{ave}} = \frac{1}{E_{div}} \left[\frac{\text{FLOPS}}{W}\right].$$

- *The author is with the Department of Applied Mathematics and Computer Science (DTU Compute), Technical Univ. of Denmark, Lyngby, Denmark. E-mail: alna@dtu.dk.*

Manuscript received 23 Mar. 2015; revised 8 June 2015; accepted 10 June 2015. Date of publication 21 June 2015; date of current version 13 Apr. 2016.
Recommended for acceptance by J.D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2448097

Fig. 1 shows the PpW trends for binary64 (double-precision) division units for different clock rates. The data are derived from [8] for a radix-4 division unit similar to that of the ARM processor [10] and a radix-16 unit similar to the one of Intel's Core processors. The clock period used to clock the unit is expressed in fanout-of-four (FO4) delay units.

From Fig. 1 we can see that the performance–per–watt curve drops from about 2.8 GFLOPS/W to about 1.7 GFLOPS/W at FO4=25 that corresponds to the minimum clock period the radix-16 unit can be clocked.

In this work, we try to keep the PpW curve as high as possible by designing a different divider architecture to fill the performance–per–watt gap for FO4 < 25. The solution is a radix-8 digit-recurrence division with minimal redundant digit set [11]. That is, three bits (radix-8 digit) of the quotient computed per iteration and digit set $[-4, 4]$.

Radix-8 division was already presented in the past in [12] and [13], and the UltraSPARC-I processor implemented a radix-8 (by overlapping three radix-2 stages) divider [14].

In this work, we apply recent design techniques for digit-recurrence division to the radix-8 divider and compare the results with those of the radix-8 divider of [13]. Moreover, we update the PpW trends of Fig. 1 with the results of the proposed implementation.

The paper is organized as follows. In Section 2, we recap the main features of the digit-recurrence algorithm and present the design of the selection function (SEL) for the minimally redundant radix-8 case. In Section 3, we detail the architecture of the divider. In Section 4, we present the results of the synthesis for the proposed radix-8 divider and we compare its performance with the radix-8 divider of [13]. In Section 5, we compare the radix-8 divider to competing units, and in Section 6, we draw the conclusions.

2 Algorithm

The division $q = \frac{x}{d}$ in the radix-8 digit-recurrence algorithm is implemented by the residual recurrence [11]:

$$w[j+1] = rw[j] - q_{j+1}d \qquad j = 0, 1, \ldots, m-1, \tag{1}$$

where $r = 8$, $w[0]$ is initialized with the dividend $x \in [0.5, 1)$ (eventually shifted to guarantee convergence), $d \in [0.5, 1)$ is the divisor, and q_{j+1} is the radix-8 quotient digit obtained by a selection function

$$q_{j+1} = SEL(d_\delta, rw[j]_t). \tag{2}$$

In (2), d_δ and $rw[j]_t$ are approximations of d and $rw[j]$ truncated after δ and t fractional bits, respectively. The number of iterations m depends on the required precision. For example, as the quotient bits produced per iteration are 3, for *binary64* (double-precision) $m = 18$ ($3 \times 18 = 54$ bits).

The quotient digit is in signed-digit representation:

$$q_{j+1} \in \{-a, \ldots, -1, 0, 1, \ldots, a\}$$

with a redundancy $\rho = \frac{a}{r-1} = a/7$. The quotient q, in sign-and-magnitude, is obtained by converting the quotient digits on-the-fly [11].

The algorithm convergence condition is

$$|w[j]| \leq \rho d, \tag{3}$$

and depending on the redundancy ρ the initialization must be done by scaling x so that (3) is not violated.

To avoid a time consuming carry-propagate addition (subtraction) in (1), the recurrence is implemented carry-save: $w_s[j], w_c[j]$. Therefore, the input $rw[j]_t$ (y in the following) to the selection function is carry-save as well:

$$y = 8w[j]_t = 8w_s[j]_t + 8w_c[j]_t = y_s + y_c.$$

0018-9340 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

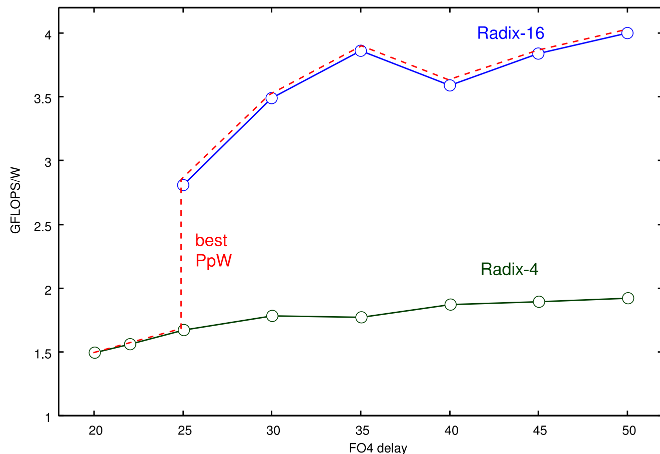

Fig. 1. Performance-per-watt trends for radix-16 and radix-4 binary64 division.

The redundancy ρ determines the complexity of the selection function in terms of δ and t bits necessary to address the table. Because d is normalized in $[0.5, 1)$ the first fractional bit d_1 is always 1, and, therefore, $\delta - 1$ fractional bits are necessary to address the table. As for y, in addition to t fractional bits, a number of integer bits (also depending on ρ) are required.

In [13], the authors made an exploration of possible redundant digit sets for radix-8. The results for the selection function are reported in Table 1. The numbers of bits necessary to address the quotient digit table increases as the redundancy decreases. However, based on delay/area tradeoffs, the authors of [13] chose $a = 7$ for their design.

In our minimally redundant implementation ($a = 4$) the resulting number of bits, refer to [1] for detail, is 5 bits for d ($\delta = 6$) and 9 bits for y ($t = 6$, plus 3 integer bits). Implementing a 2^{5+9} =16K entry table as in the approach of [13] is prohibitively expensive for $a = 4$.

However, by implementing *"selection by comparison"* ([10], [15]) the table cost is greatly reduced. As the bits of d (d_δ) do not change across the iterations, we can just store the constant thresholds (m_k) in the table, and compare them to y, i.e. SIGN$(y - m_k)$, to determine the quotient digit. With this choice, the $a = 4$ selection constants table is reduced to $2^5 = 32$ entries and the comparison is done by several 10-bit subtractions in parallel. For the $a = 7$ selection function of [13], selection by comparison would result in a 8-entry constants table plus 9-bit adders. Therefore, the delays would be similar.

Once a is decided, the redundancy is $\rho = 4/7$ and the condition of convergence (3) on the initial residual is

$$w[0] \leq \frac{4}{7} \cdot d.$$

The worst case is for $w[0] = x = 1.0 - 2^{-53} = 0.111\ldots11$ and $d = 0.5$, that is,

$$0.111\ldots11 \leq \frac{4}{7} \cdot 0.5 = 0.010010\ldots.$$

Therefore, to ensure convergence in the first iteration, x must be divided by 4 (shift right two positions) $w[0] = x/4$. Because due to shifting, the first quotient digit might be zero, it is necessary to run an extra iteration to have a *binary64* quotient (significand).

TABLE 1
Radix-8 Selection Function Exploration from [13]

a	n. bits d	n. bits y	delay [ns]	area [NAND2]
10	3	6	4.0	325
7	3	8	3.8	370
6	5	7	3.8	420

TABLE 2
Selection Constants for Radix-8, $a = 4$ Division

d_δ	m_4	m_3	m_2	m_1	m_0	m_{-1}	m_{-2}	m_{-3}
0.100000	113 114	81	48	16	-16	-48	-81	**-114** **-115**
0.100001	116 117	83	50	16	-16	-50	-83	**-117** **-118**
0.100010	120	86	52	16	-16	-52	-86	**-121**
0.100011	124	88	52	16	-16	-52	-88	-124
0.100100	128	91	54	18	-18	-54	-91	-128
0.100101	131	93	56	18	-18	-56	-93	-131
0.100110	135	96	58	18	-18	-58	-96	-135
0.100111	138	98	58	18	-18	-58	-98	-138
0.101000	141	101	60	20	-20	-60	-101	-141
0.101001	145	103	62	20	-20	-62	-103	-145
0.101010	148	105	62	20	-20	-62	-105	-148
0.101011	152	108	64	20	-20	-64	-108	-152
0.101100	155	110	66	20	-20	-66	-110	-155
0.101101	159	113	68	24	-24	-68	-113	-159
0.101110	162	115	68	24	-24	-68	-115	-162
0.101111	166	118	70	24	-24	-70	-118	-166
0.110000	169	120	72	24	-24	-72	-120	-169
0.110001	172	123	72	24	-24	-72	-123	-172
0.110010	176	125	76	24	-24	-76	-125	-176
0.110011	179	128	76	24	-24	-76	-128	-179
0.110100	184	130	78	24	-24	-78	-130	-184
0.110101	186	133	78	24	-24	-78	-133	-186
0.110110	190	135	82	28	-28	-82	-135	-190
0.110111	193	138	82	28	-28	-82	-138	-193
0.111000	197	140	86	28	-28	-86	-140	-197
0.111001	200	142	86	28	-28	-86	-142	-200
0.111010	204	145	86	28	-28	-86	-145	-204
0.111011	207	147	88	28	-28	-88	-147	-207
0.111100	211	150	90	28	-28	-90	-150	-211
0.111101	214	152	90	28	-28	-90	-152	-214
0.111110	218	155	94	28	-28	-94	-155	-218
0.111111	221	158	94	28	-28	-94	-158	-221

Values m_k's are multiplied by 64=2^6.

Summarizing, for $r = 8$ and $a = 4$, the required iterations for *binary64* in (1) are $m = 19$.

2.1 Selection Function

By choosing $a = 4$, the radix-8 quotient digit can be

$$q_{j+1} \in \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$$

and it is determined by performing a comparison [10] of the truncated residual y with the eight values m_k representing the boundaries to select the digit for the given d. That is,

$$\begin{array}{rcccccl} & & y & \geq & m_4 & \rightarrow & q_{j+1} = 4 \\ m_3 & \leq & y & < & m_4 & \rightarrow & q_{j+1} = 3 \\ m_2 & \leq & y & < & m_3 & \rightarrow & q_{j+1} = 2 \\ m_1 & \leq & y & < & m_2 & \rightarrow & q_{j+1} = 1 \\ m_0 & \leq & y & < & m_1 & \rightarrow & q_{j+1} = 0 \\ m_{-1} & \leq & y & < & m_0 & \rightarrow & q_{j+1} = -1 \\ m_{-2} & \leq & y & < & m_{-1} & \rightarrow & q_{j+1} = -2 \\ m_{-3} & \leq & y & < & m_{-2} & \rightarrow & q_{j+1} = -3 \\ & & y & < & m_{-3} & \rightarrow & q_{j+1} = -4. \end{array} \quad (4)$$

To determine the constants m_k for each interval of d, we have to design the selection function by selecting values of δ and t to ensure the convergence of the algorithm [1].

For $r = 8$ and $a = 4$, we chose $\delta = 6$, corresponding to $2^5 = 32$ intervals on d, $t = 6$ fractional bits plus 3 integer bits for y. The selection constants table is reported in Table 2.

By implementing the constants table by a ROM[1] its size is $32 \times (8 \times 9) = 2{,}304$ bits.

1. Actually, the table is synthesized in multi-level combinational logic.

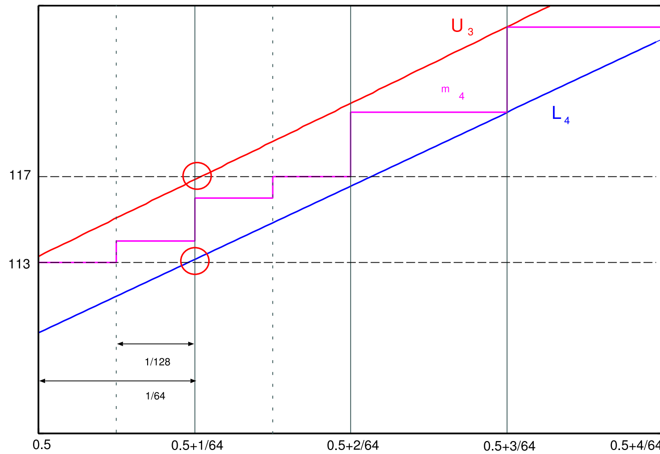

Fig. 2. PD plot for m_4 in interval $d \in [0.5, 0.5 + \frac{4}{64})$. The selection constants must lie in the region between the upper bound to select $q_{j+1} = 3$ (U_3) and the lower bound to select $q_{j+1} = 4$ (L_4). The red circles indicate the out-of-bound errors for intervals of width 1/64.

However, to reduce the constants table size, we can exploit the symmetries in the selection function, with respect to the positive/negative values of y, by storing only the positive constant and obtaining the negative constant by two's complement:

$$m_{-3} = -m_4, \; m_{-2} = -m_3, \; m_{-1} = -m_2, \; m_0 = -m_1.$$

In this way, the ROM size is halved: $32 \times (4 \times 9) = 1,152$ bits.

For a few values when $d \in [0.5, 0.5 + \frac{3}{64})$ (first three intervals) the constant $m_{-3} = -m_4$ produces out-of-bound $w[j]$ in (3). However the problem can be easily overcome by taking the one's complement of m_4:

$$m_{-3} = -m_4 - 1.$$

An extra bit for each row of the ROM is required to signal those cases (boldface in Table 2).

The selection constants are verified analytically against the bounds in each interval on d, and in the first two intervals $d \in [0.5, 0.5 + \frac{2}{64})$ the constant m_4 resulted out-of-bound for values close to $d = 0.5 + \frac{1}{64}$. This is graphically illustrated in the plot of Fig. 2 where the divisor d is shown on the x-axis, and the shifted residual $rw[j]$ on the y-axis. This type of plot, is referred as PD plot, or PD diagram [11].

This error can be fixed by increasing the number of intervals on d. However, this solution is quite expensive and because the error only occurs in the first two intervals, an easy fix is to use the bit of d of weight 2^{-7} (called b) to select the sub-interval in these two cases only. This is implemented by truncating the two least-significant bits of m_k and by completing the constant with b, as shown in Table 3.

3 Radix-8 Divider Architecture

In the following, we describe the architecture to implement the *binary64* significand computation of radix-8 division. We omit the description of how to obtain exponent and sign as it is trivial for division.

TABLE 3
Implementation of m_4 in First Two d Intervals

d_δ	b	m_4 (binary)	m_4 (decimal)
0.100000	0	$0011100b\bar{b}$	113
	1		114
0.100001	0	$00111010b$	116
	1		117

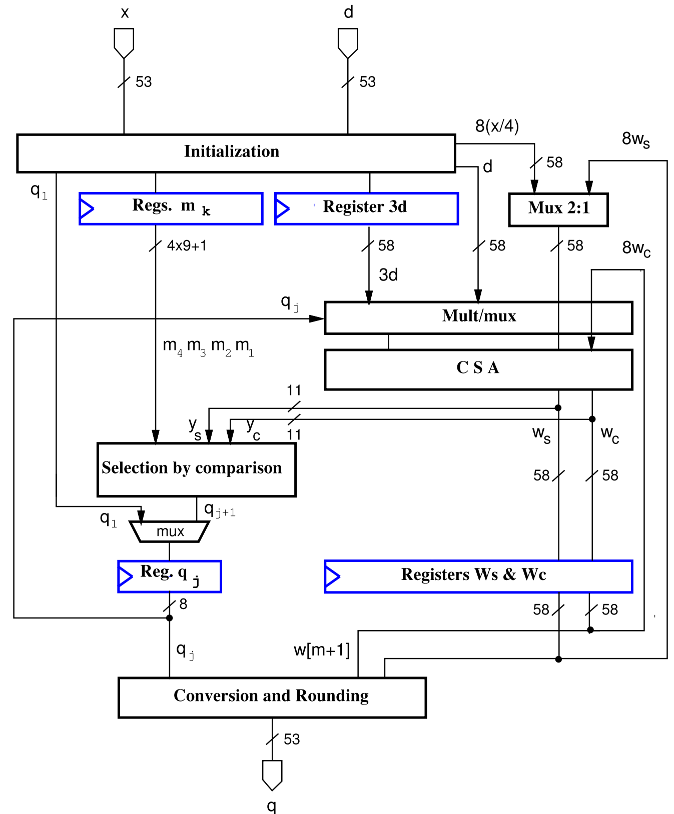

Fig. 3. Architecture of radix-8 division unit (significand).

The first design choice to make is how to obtain the multiples $-q_{j+1}d$ in each iteration (1). For $a = 4$ all multiples are simple (can be obtained by shifts and/or two's complement) except 3. To compute $3d$, we have two choices:

1) to add d and $2d$ to the carry-save representation of w;
2) to precompute $3d = 2d + d$ with a carry-propagate adder (CPA).

By choice 1, we have to use a 4:2 carry-save adder (CSA) in the recurrence (1), which is slower than a 3:2 CSA, and, therefore, increase the cycle time.

By choice 2, we can compute $3d$ at initialization, store it in a register, and implement the recurrence (1) with a 3:2 CSA. Clearly, a 56-bit CPA plus a register are required, but if this addition is the only operation in the cycle, it can be done fast enough not to exceed the recurrence cycle time.

By retiming the recurrence of (1), that is the quotient digit q_{j+1} is computed at the end of the cycle, we limit to a minimum the bits of the residual $w[j]$ in the critical path [13]. The retimed recurrence is:

$$w[j] = rw[j-1] - q_j d \qquad j = 1, \ldots, m \tag{5}$$

with $q_0 = 0$ and selection function

$$q_{j+1} = SEL(d_\delta, w[j]_t). \tag{6}$$

Since q_{j+1} is used to compute $w[j]$ in the next cycle, it has to be stored in a register.

The architecture we chose to implement the radix-8 ($a = 4$) division is shown in Fig. 3. The unit is completed by a control unit and control signals not depicted in the figure. The detail of the selection by comparison, implemented by (4), is shown in Fig. 4. The quotient digit q_j is one-hot encoded with 8 bits (e.g., $q_j = 0 \rightarrow 00000000$, $q_j = 4 \rightarrow 10000000$, etc.).

In the *Initialization* (first cycle), we perform the following operations:

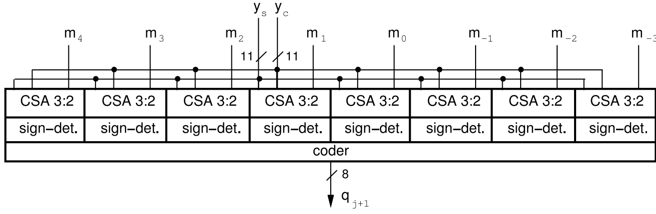

Fig. 4. Implementation of selection by comparison.

1) We precompute $3d = 2d + d$ by a CPA and store it at the end of the cycle.
2) We address the constant table with d_δ and store the four constants m_4–m_1 at the end of the cycle.

However, since at initialization $w[0] = x/4$, we can compute the first quotient-digit q_1 by a simplified selection function as explained next.

In the first iteration $rw[0] = 8 \cdot (x/4)$, and for the normalization of x:

$$rw[0] \in [1, 2) \to 64 \leq y < 128$$

(bounds on y are multiplied by 2^6 as in Table 2).

Therefore, the selection function in the first iteration is limited to the space in the PD plot shown in Fig. 5. In each interval on d (x-axis), we have at most three possible values of q_1, depending on the value of $y = 2x$ (y-axis).

Because in the first cycle we obtain the constants m_k's from the table, we use two of them to determine q_1. From Fig. 5, we notice that only the first five d_δ intervals can have $q_1 = 4$. By marking these intervals in the m_k's table with an extra bit f set to $f = 1$, we can select the right constants for each interval (m_H and m_L) and determine q_1 as:

$$\begin{array}{rcrcrcl} & & y & \geq & m_H & \rightarrow & q_1 = 3 + f, \\ m_L & \leq & y & < & m_H & \rightarrow & q_1 = 2 + f, \\ & & y & < & m_L & \rightarrow & q_1 = 1 + f. \end{array} \quad (7)$$

The detail of the implementation of the initialization cycle is shown in Fig. 6.

By implementing q_1 in the first cycle, we save an iteration in the recurrence. However, since q_1 must be stored in the quotient digit register, we need to add a 2:1 multiplexer that is in the critical path as detailed in Section 4.

The divider is completed by a convert-and-round unit which converts the quotient digits from signed-digit to (unsigned) magnitude as q_j's are produced [1]. The rounding is done by checking the sign of the remainder $w[m + 1]$ and if it is zero.

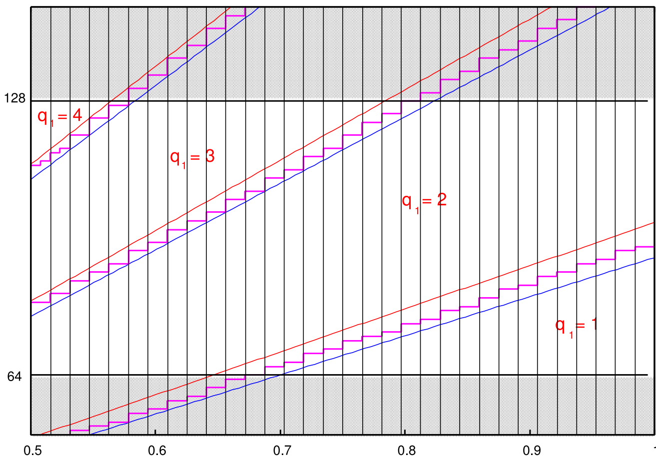

Fig. 5. Quotient digit selection space in the first iteration.

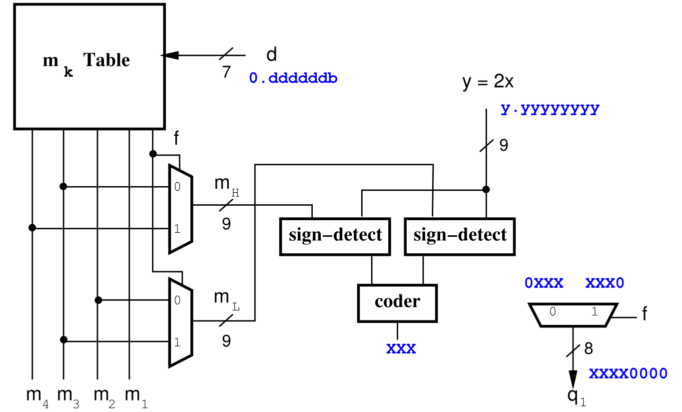

Fig. 6. Quotient digit selection q_1 in first cycle (part of *Initialization* block).

In the retimed implementation, the total number of iterations is 22.

- **Initialization** (cycle 1):
  The constants m_k are retrieved, q_1 is computed, and $3d$ is precomputed.
- **Recurrence** (cycles 2-20):
  In each cycle q_{j+1} and $w[j]$ are computed in the recurrence. In the convert-and-round unit, digits are converted with one extra cycle delay (e.g., q_1 is converted in cycle 3).
- **Remainder** (cycle 21):
  At the end of cycle 20, all quotient digits are computed. In cycle 21, the last digit q_{19} is converted and the sign of the remainder (and if it is zero) is computed.
- **Rounding** (cycle 22): Rounding is performed.

3.1 Testing

To test the architecture of Fig. 3, we created a cycle-accurate and bit-accurate model in the C language and we ran simulations on billions of pairs of *binary64* significands to check for out-of-bounds errors and for correct rounding.

The fraction of the *binary64* significands[2] (52 bits) was divided into an upper part f_U of 16 bits and a lower part f_T of 36 bits. For f_U, we generated all possible combinations of bits for dividend x and divisor d for a total of $2^{16} \times 2^{16} = 2^{32}$ combinations. The lower part f_T of x and d is filled with different patterns: random generated bits, $f_T = 0$ (all bits set to 0), and $f_T = 2^{36} - 1$ (all bits set to 1).

We ran several sets of 2^{32}-pattern simulations, by generating all combinations for f_U and specific patterns for f_T. For example:

	x_T	d_T
set 1:	random	random
set 2:	0	$2^{36} - 1$
...	...	...

The running time for each set of 2^{32} simulations is about 130 hours ($5\frac{1}{2}$ days) per core on a mini-server.

4 Radix-8 Divider Synthesis

The unit is coded in VHDL at RTL-level and synthesized by Synopsys's Design Compiler in the STM 90 nm CMOS standard cell library. The library is not the latest available, but it is the one used in [8] and this makes the comparisons straightforward. Moreover, it is a modern library with three types of cells for low power design (standard, high and low V_t threshold voltages) which guarantees results are scalable to smaller features (e.g., 45 or 32 nm). For

2. We do not consider subnormals in this test.

TABLE 4
Synthesis Results for the Radix-8 $a = 4$ Divider

radix-8 ($a = 4$) divider (22 cycles)						
T_C FO4	T_C $[ps]$	Area NAND2	Latency $[ns]$	P_{ave} $[mW]$	E_{div} $[pJ]$	PpW
50	2,250	5,786	49.5	1.33	264	3.78
45	2,025	5,936	44.6	1.38	274	3.64
40	1,800	6,224	39.6	1.42	283	3.53
35	1,575	6,584	34.7	1.49	299	3.34
30	1,350	6,809	29.7	1.54	310	3.22
25	1,125	6,957	24.8	1.55	312	3.20
22	990	7,493	21.8	1.77	361	2.77
20	900	8,266	19.8	2.06	424	2.36

P_{ave} estimated at 100 MHz.
PpW is performance-per-watt in GFLOPS/W.

comparison purposes, the FO4 inverter delay is $45ps$ and the area of the NAND2 gate is $4.4\mu m^2$ in the STM 90 nm CMOS library.

The synthesis was done under different timing constraints (T_C) ranging from 50 FO4 (2,250 ps) to 18 FO4 (810 ps) to determine the minimum clock cycle and to see how area and power dissipation scale with timing.

We estimated the power dissipation by using Synopsys's Power Compiler based on the switching activity obtained by a Monte Carlo simulation (random vectors). For all the synthesis runs, the average power dissipation was estimated at a normalized frequency of 100 MHz.

We report the synthesis results in Table 4. The fastest implementation is obtained for FO4=20 ($T_C = 900$ ps). The delay of the critical path is

$$\begin{array}{ccccccccccccc} \text{reg.}q_j & & \text{Mult} & & \text{CSA} & & \text{SEL} & & \text{mux}q_j & & \text{(set-up)} & & \\ 105 & + & 259 & + & 66 & + & 344 & + & 40 & + & 86 & = & 900 \text{ ps.} \end{array}$$

We synthesize in the same 90 nm library, the radix-8 $a = 7$ divider of [13] under the same set of timing constraints to determine whether the minimally redundant radix-8 ($a = 4$) is more power efficient.

The unit of [13] is shown in Fig. 7. Differently from the $a = 4$ divider, the quotient-digit is split in two parts such as $q_j = 4q_{Hj} + q_{Lj}$, and, consequently, two wide (56 bits) muxes and CSAs are required in the recurrence. Moreover, the selection function is not implemented by comparison, but the y and d_δ address the quotient-digit table, synthesized as multi-level combinational logic.

The radix-8 $a = 7$ unit fails to meet the timing constraint for FO4=22 (990ps):

$$\begin{array}{ccccccccccccc} \text{reg.}q_H & & \text{Mult} & & \text{CSA} & & \text{CSA} & & \text{SEL} & & \text{(set-up)} & & \\ 111 & + & 151 & + & 65 & + & 112 & + & 472 & + & 86 & = & 997 \text{ ps.} \end{array}$$

The radix-8 $a = 4$ divider is slightly faster (2 FO4) than the radix-8 $a = 7$. Its selection function is faster (selection by comparison) and there is only one CSA in the recurrence path. However, the multiple generator for $a = 4$ is more complicated, and slower, since the selection is among nine multiples, while for $a = 7$ the selection is among five multiples in each multiple generator, and only one multiple generator is on the critical path.

The synthesis results for the two radix-8 dividers, for area and power dissipation, are reported for the FO4=22 implementation (the fastest comparable) in Table 5. The multiple generators, the CSAs, and the registers to store $w[j]$ are grouped in "w-path" in the table.

Table 5 shows that the selection function block in $a = 4$ consumes significantly more power than the SEL in $a = 7$. The initialization block (CPA $3d$, m_k Table and selection q_1) takes more than 20 percent the area, but the switching is reduced to one cycle and the power dissipation contributes to less than 5 percent of the total. As a result, the area of the radix-8 $a = 4$ divider is about 20 percent

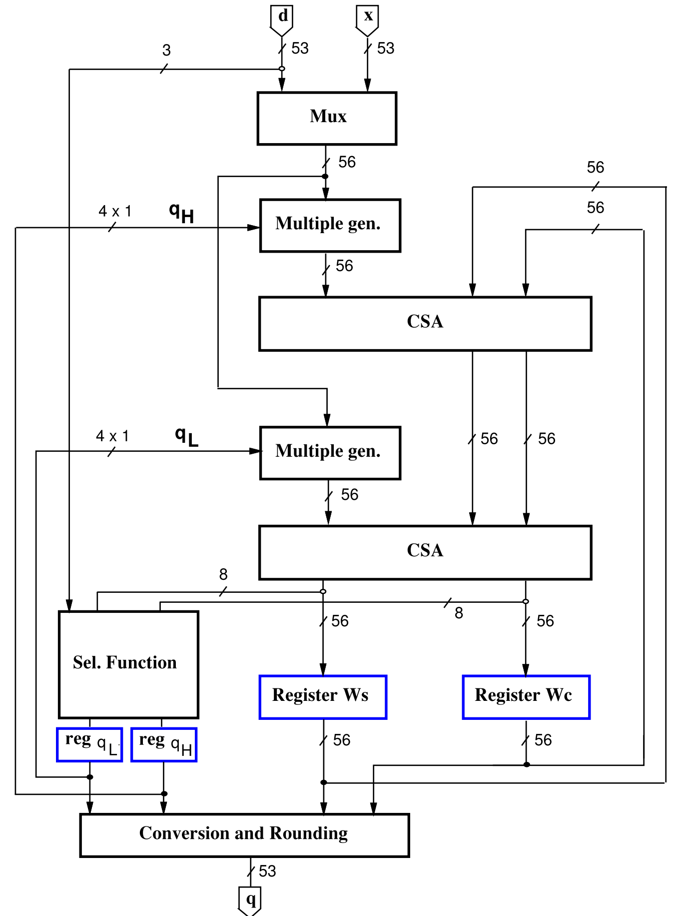

Fig. 7. Radix-8 $a = 7$ implementation from [13].

larger than that of $a = 7$, but the average power dissipation is slightly smaller for $a = 4$.

Summarizing, the radix-8 $a = 4$ divider can be clocked faster than the radix-8 $a = 7$ divider (20 versus 22 FO4 delay), and the average power dissipation, and energy-per-division, is slightly smaller.

5 Comparison to Radix-4 and Radix-16

We compared the radix-8 $a = 4$ unit to a radix-4 unit similar to that of [10], and to the radix-16 unit of [8].

In the radix-4 ($a = 2$) unit of [10] the quotient-digit q_j is determined by performing a comparison of the truncated residual y with the four selection constants (m_k). In parallel, all five possible residuals $w^k[j+1]$ are computed speculatively, and then, once q_j is determined, the corresponding residual is selected.

TABLE 5
Area and Power Breakdown for Radix-8 Dividers $a = 4$ and $a = 7$ at FO4=22

	radix-8 $a = 4$				radix-8 $a = 7$			
	Area		P_{ave}		Area		P_{ave}	
block	NAND2	%	$[mW]$	%	NAND2	%	$[mW]$	%
Init.	1,655	22	0.07	4	–		–	
Reg. m_k	161	2	0.04	3	–		–	
SEL	1,469	20	0.54	32	905	14	0.25	15
Reg. q_j	39	1	0.01	1	34	1	0.01	1
w-path	2,439	33	0.67	39	3,499	56	1.08	63
C&R	1,731	23	0.35	21	1,836	29	0.38	22
Total	7,493	100	1.69	100	6,275	100	1.72	100

TABLE 6
Comparison Radix-4, Radix-8 ($a = 4$) and Radix-16 Dividers

T_C FO4	Throughput [$MFLOPS$]			P_{ave} at 100 MHz [mW]			Energy–per–division [pJ]			Performance–per-watt [$GFLOPS/W$]		
	radix-4	radix-8	radix-16	radix-4	radix-8	radix-16	radix-4	radix-8	radix-16	radix-4	radix-8	radix-16
50	14.81	20.20	27.78	1.71	1.33	1.56	512	264	250	1.95	3.78	**4.00**
45	16.46	22.45	30.86	1.73	1.38	1.63	520	274	261	1.92	3.64	**3.84**
40	18.52	25.25	34.72	1.75	1.42	1.74	526	283	278	1.90	3.53	**3.59**
35	21.16	28.86	39.68	1.85	1.49	1.62	555	299	259	1.80	3.34	**3.86**
30	24.69	33.67	46.30	1.84	1.54	1.79	552	310	287	1.81	3.22	**3.49**
25	29.63	40.40	55.56	1.96	1.55	2.23	587	312	356	1.70	**3.20**	2.81
22	33.67	45.91	–	2.10	1.77	–	630	361	–	1.59	**2.77**	–
20	37.04	50.51	–	2.19	2.06	–	657	424	–	1.52	**2.36**	–

"-"means that the unit cannot meet the timing constraint. The best PpW per timing constraint T_C is in boldface.

The architecture of [10] was designed to guarantee a very reduced logical depth (i.e., high clock rate). However, the speculation on $w[j+1]$ in the recurrence (four 56-bit CSAs in parallel) makes the unit quite power hungry.

The fastest implementation obtained for the synthesis of the radix-4 divider is 860 ps corresponding to about 19 FO4. This value is quite close to the delay of 17 FO4 reported in [10].

As for the radix-16 division unit, the implementation of [8] is done in the same library. Therefore, we used the data provided in [8] for the comparison. The radix-16 unit of [8], is obtained by overlapping two radix-4 stages and has a latency of 16 clock cycles.

Table 6 summarizes throughput, average power dissipation, energy-per-division, and performance-per-watt (PpW), derived from the synthesis results for the radix-4, radix-8 ($a = 4$), and radix-16 dividers.

By comparing the radix-8 and the radix-16 dividers, E_{div} in Table 6 is similar for the two units, while the throughput is higher for radix-16 because the latency (number of cycles) is shorter than the radix-8.

By comparing the PpW between radix-8 and radix-16, for relaxed timing constraints (FO4 $>$ 30) the FLOPS "weight" more than the power P_{ave}, but when the units operate close to the maximum speed, the power increases at a higher rate than FLOPS. At FO4=25, minimum clock cycle period for radix-16, the radix-8 is more energy efficient than the radix-16 divider.

The boldface values in Table 6 indicate the best PpW at the corresponding synthesis clock period (FO4 delay). From clock rates up to $\frac{1}{30FO4}$ (740 MHz in our library) the radix-16 divider is the one with the best PpW, while for clock rates between $\frac{1}{25FO4}$ and $\frac{1}{20FO4}$ (900-1,100 MHz) the radix-8 divider gives the best PpW.

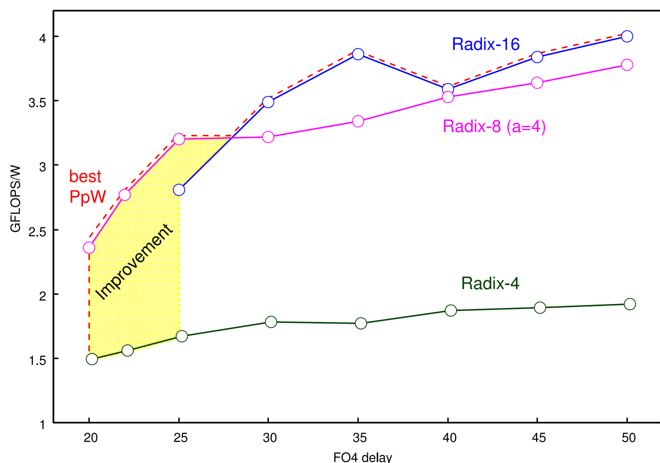

Fig. 8. Performance/watt trends for radix-4, radix-8 and radix-16 binary64 division.

Fig. 8 shows the improvement over Fig. 1 (Section 1) in the best PpW obtained by adding the radix-8 divider in the performance–per–watt space.

6 Conclusions

The main contribution of this work is the development of a minimally redundant radix-8 divider designed to achieve good energy efficiency.

With respect to previous work on radix-8 division, the radix-8 ($a = 4$) divider is more compact (one multiple generator and one CSA) in the recurrence at expenses of a more complicated quotient-digit selection.

The main challenge in the design is to simplify the selection function, and to achieve this we resorted to known techniques such as selection by comparison and choosing symmetrical constants to reduce the constants table size.

We also provided novel solutions, such as conditionally splitting some selection intervals to not double their number, and the computation of the first quotient-digit in the first cycle by a simplified selection function.

The results of the synthesized minimally redundant radix-8 divider compare favorably to those of a radix-8 $a = 7$ unit since the proposed unit can be clocked faster and consumes less energy.

With respect to the radix-16 divider, the radix-8 unit can be clocked faster. For high clock rates, the radix-8 divider completes a division by consuming less energy than the radix-16 unit because its average power dissipation is lower and its longer latency does not offset the savings.

Moreover, from FO4=25 and below, the radix-8 unit is the most energy efficient in terms of FLOPS per watt.

References

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

[2] H. Baliga, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles. (2008, Oct.). Improvements in the Intel Core2 Penryn processor family architecture and microarchitecture. *Intel Technol. J.*, pp. 179–192 [Online]. Available: http://www.intel.com/technology/itj/2008/v12i3/3-paper/1-abstract.htm

[3] N. Burgess and C. N. Hinds, "Design of the ARM VFP11 divide and square root synthesisable macrocell," in *Proc. 18th IEEE Symp. Comput. Arithmetic*, Jul. 2007, pp. 87–96.

[4] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess, "High performance floating-point unit with 116 bit wide divider," in *Proc. 16th Symp. Comput. Arithmetic*, 2003, pp. 87–94.

[5] S. F. Oberman, "Floating-point division and square root algorithms and implementation in the AMD-K7 microprocessor," in *Proc. 14th Symp. Comput. Arithmetic*, 1999, pp. 106–115.

[6] NVIDIA, "Fermi. NVIDIA's Next Generation CUDA Compute Architecture," Whitepaper [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[7] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, Sep./Oct. 2000.

[8] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1059–1070, Aug. 2012.
[9] M. B. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, Sept./Oct. 2013.
[10] N. Burgess and C. Hinds, "Design issues in radix-4 SRT square root and divide unit," in *Proc. 35th Asilomar Conf. Signals, Syst. Comput.*, 2001, pp. 1646–1650.
[11] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Norwell, MA, USA: Kluwer, 1994.
[12] J. Fandrianto, "Algorithm for high-speed shared radix-8 division and radix-8 square root," in *Proc. 9th Symp. Comput. Arithmetic*, Sep. 1989, pp. 68–75.
[13] A. Nannarelli and T. Lang, "Low-power radix-8 divider," in *Proc. Int. Conf. Comput. Des.*, Oct. 1998, pp. 420–426.
[14] A. Prabhu and G. Zyner, "167 MHz radix-8 divide and square root using overlapped radix-2 stages," in *Proc. 12th Symp. Comput. Arithmetic*, July. 1995, pp. 155–162.
[15] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli, "Digit-recurrence dividers with reduced logical depth," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 837–851, July. 2005.