Design and Implementation of a Radix-4 Complex Division Unit with Prescaling
Pouya Dormiani, Milos Ercegovac, Jean-Michel Muller

To cite this version:

HAL Id: ensl-00379147
https://hal-ens-lyon.archives-ouvertes.fr/ensl-00379147v2
Submitted on 13 Jan 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Design and Implementation of a Radix-4 Complex Division Unit with Prescaling

Pouya Dormiani
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024, USA
Email: pouya@cs.ucla.edu

Miloš D. Ercegovac
4731H Boelter Hall
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024, USA
Email: milos@cs.ucla.edu

Jean-Michel Muller
CNRS-Laboratoire CNRS-ENSL-INRIA-UCBL LIP
Ecole Normale Supérieure de Lyon
46 Allée d’Italie
69364 Lyon Cedex 07, France
Email: Jean-Michel.Muller@ens-lyon.fr

Abstract—We present a design and implementation of a radix-4 complex division unit with prescaling of the operands. Specifically, we extend the treatment of the residual bound and errors due to the use of truncated redundant representation. The requirements for prescaling tables are simplified and a detailed specification of the table design is given. All principal components used in the design are described and the proposed optimizations are explained. The target platform for implementation was an Altera Stratix II FPGA [15] for which we report timing and area requirements. For a precision of 36 bits, the implementation uses 1185 ALUTs, achieving a latency of 157 ns. The maximum clock frequency is 173.49 MHz.

I. INTRODUCTION

Complex division is used in applications such as signal processing (e.g., the complex SVD), multiantenna systems (MIMO-type) [1], GPS [2], astronomy [3], and non-linear RF measurement [4]. Unlike for complex multipliers [10], [12], its implementation has been commonly provided in software. If we want to improve its performance, a hardware implementation is considered. With that objective, a hardware-oriented algorithm and the corresponding theory for general radix-4 complex valued division based on a digit-recurrence algorithm has been introduced in [6]. A high-level design of a complex divider is discussed in [7] without implementation details. In this paper we focus on the design and implementation of a radix-4 complex-valued division unit with the quotient-digit selection in the complex domain is a two-dimensional problem which can be computed in parallel:

\begin{align}
 w^R[j + 1] &= 4w^R[j] - q_{j+1}^Rd^R + q_{j+1}^Id^I \\
 w^I[j + 1] &= 4w^I[j] - q_{j+1}^Rd^I - q_{j+1}^Id^R
\end{align}

where \( w^R[0] = z^R \) and \( w^I[0] = z^I \). The quotient-digit selection in the complex domain is a two-dimensional problem because both \( q_{j+1}^R \) and \( q_{j+1}^I \) must be selected in such a way that the real and imaginary residuals \( (w^R[j], w^I[j]) \) remain bounded. This is much more difficult than single-digit selection used in the real case. We solve this problem by scaling the operands by factor \( K \) such that \( Kz/Kd = x/y \). The quotient-digit selection in the complex domain is a two-dimensional problem because both \( q_{j+1}^R \) and \( q_{j+1}^I \) must be selected in such a way that the real and imaginary residuals \( (w^R[j], w^I[j]) \) remain bounded. This is much more difficult than single-digit selection used in the real case. We solve this problem by scaling the operands by factor \( K \) such that \( Kz/Kd = x/y \) where \( y = Kd \approx 1 \). Consequently, \( y^R \approx 1 \) and \( y^I \approx 0 \), and the selection of \( q_{j+1}^R \) and \( q_{j+1}^I \) can be performed on the real and imaginary shifted residuals separately in a manner similar to real-valued division selection. To determine the prescaling factor \( K \), we assume that

\[ \|Kd - 1\|_\infty < \epsilon_s \]

where \( \|a\|_\infty = \max(\|a^R\|, \|a^I\|) \).

After prescaling the recurrences are

\begin{align}
 w^R[j + 1] &= 4w^R[j] - q_{j+1}^Ry^R + q_{j+1}^Iy^I \\
 w^I[j + 1] &= 4w^I[j] - q_{j+1}^Ry^I - q_{j+1}^Iy^R
\end{align}

where \( w^R[0] = x^R \) and \( w^I[0] = x^I \). Because the scaling makes \( y^I \approx 0 \) and \( y^R \approx 1 - \epsilon_s \), the selection of the real part of the quotient can be performed by rounding the shifted real residual and taking its integer part. Similarly for the selection of the imaginary part of the quotient digit. Moreover, we can use estimates with \( \sigma \) fractional positions of the shifted residuals \( 4w^R[j] \) and \( 4w^I[j] \) in the selection. Consequently,
the residuals can be computed in redundant form to keep the cycle time short. The selection functions are

\[ q^R_{j+1} = \text{Sel}(\text{est}(4w^R[j], \sigma)) \]
\[ = \text{sign}(4w^R[j]) \times \|\text{est}(4w^R[j], \sigma)\| + \frac{1}{2} \]
\[ q^I_{j+1} = \text{Sel}(4w^I[j], \sigma)) \]
\[ = \text{sign}(4w^I[j]) \times \|\text{est}(4w^I[j], \sigma)\| + \frac{1}{2} \]

The selection function \(\text{Sel}\) satisfies

\[ |\text{Sel}(\text{est}(x, \sigma)) - x| < \frac{1}{2} + 2^{-\sigma} \] (12)

The \(\text{est}(x, \sigma)\) is \(x\) truncated to \(\sigma\) fractional positions with an error bound

\[ \text{est}_{ERR}(x, \sigma) = |x - \text{est}(x, \sigma)| < 2^{-\sigma} \]

If \(x\) is in carry-save form \(x = x_C + x_S\) then truncating the carry and sum vector to \(\sigma + 1\) fractional bits results in the same maximum error committed, i.e., \(\text{est}_{ERR}(x, \sigma) < 2^{-\sigma}\) and \(\text{est}_{ERR}(x_C, \sigma + 1) + \text{est}_{ERR}(x_S, \sigma + 1) < 2^{-\sigma}\).

Using (10), (11) and (12), a bound on the residual is deduced which ensures that the digit \((q^R_{j+1}, q^I_{j+1})\) selected by rounding is in the digit set \([-3, \ldots, 3]\). Namely,

\[ \|w[j]\|_\infty \leq \frac{1}{4} \left( 3 + \frac{1}{2} + 2^{-\sigma} \right) \] (13)

As shown in [6], assuming that the scaling error is \(\epsilon_s\) and \(a = 3\), the residual is bounded by

\[ \|w[j]\|_\infty < 2 \times 3 \times \epsilon_s + \frac{1}{2} + 2^{-\sigma} \] (14)

Consequently,

\[ 6\epsilon_s + \frac{1}{2} + 2^{-\sigma} \leq \frac{1}{4} \left( 3 + \frac{1}{2} + 2^{-\sigma} \right) \] (15)

Satisfying this condition guarantees convergence of the digit-recurrence algorithm and allows the choice of \(\epsilon_s\) and \(\sigma\) to optimize the implementation characteristics.

II. DESIGN

The design of the complex division unit consists of several components: the prescaling module, the recurrence modules for the real and imaginary parts, the on-the-fly converters to obtain conventional representations, and a simple controller. A high level block diagram of the design is shown in Fig. 1 with the timing shown in Fig. 2. The prescaling module in Fig. 1 performs a ROM look-up using a short estimate of the value of the divisor \(d\) as an address, in which the ROM stores \(K = 1/d\). It then computes the complex product \(Kz\), which is used to initialize \(w[0]\) in the recurrence modules. The prescaling module computes \(Kd\) in parallel to the initialization of the recurrence modules, which is then used to perform the iterations of the recurrence. The initial delay of the module to perform prescaling can be amortized by overlapping the prescaling of the next operation with digit-recurrence iterations of the current operation—this however has not been performed in the current implementation. Detailed design of the prescaling is discussed in Section II-A.

The two recurrence modules (one for the real recurrence and one for the imaginary) perform nearly identical operations which can be mapped to the same hardware. Detailed design of the recurrence module is discussed in Section II-C.

A. Prescaling

Prescaling consists of several steps: obtaining the factor \(K\) from a table based on an short-precision estimate of \(d\), and computing \(Kz\) and \(Kd\).

We define a function \(\text{rnd}(a, b)\) which returns a rounded value of \(a\) to \(b\) fractional places, s.t. \(|a - \text{rnd}(a, b)| \leq \frac{1}{2}2^{-b}\). The factor \(K\) can be determined by using a short estimate of \(d\) to \(q\) fractional positions, i.e., \(\text{rnd}(d^R, q), \text{rnd}(d^I, q)\) as an address to a ROM which stores the corresponding values of \(K\) with precision of \(t\) fractional positions,

\[ K^R = \text{rnd}(1/\text{rnd}(d^R, q), t) \]
\[ K^I = \text{rnd}(1/\text{rnd}(d^I, q), t) \]

Error analysis for the choices of parameters \(q\) and \(t\) is performed in [13]. These effect \(\epsilon_s\) used in (15) to guarantee convergence of the algorithm. Radix 4, with digit set \([-3, \ldots, 3]\) offers the most favorable choice of parameters by minimizing the number of bits required for the ROM among radices 4, 8,
and 16, except radix 2 which has lowest memory requirements. Over-redundant digit sets are another design choice but we decided to restrict our design to maximally redundant digit set which allows faithful rounding [6].

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>σ</th>
<th>q</th>
<th>t</th>
<th>KBits (approx.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7.5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>146</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>33</td>
</tr>
</tbody>
</table>

Radix: 8, 16 see [13] ≥ 146

TABLE I
MEMORY (ROM) REQUIREMENTS OF DIFFERENT RADIX (r), DIGIT SET \{-a, ..., a\}, PRECISION OF RESIDUAL ESTIMATE FOR SELECTION (σ), PRECISION OF d USED TO PERFORM TABLE LOOK-UP (q) AND PRECISION OF TABLE ENTRIES (t).

The value of the divisor is in the usual range
\[ \frac{1}{2} \leq \|d\|_{\infty} < 1 \] (16)
noting that larger values can be scaled to this range. Its estimate \( \text{rnd}(d, q) \) can be represented as 2 two’s complement numbers for the real and imaginary parts
\[ \text{rnd}(d, q) = \text{rnd}(d^R, q) + i \text{rnd}(d^I, q) \] (17)
\[ \text{rnd}(d^R, q) = \kappa_0 R_1 \kappa_1 R_2 \kappa_2 R_3 \ldots R_{q-1} R_q \] (18)
\[ \text{rnd}(d^I, q) = \kappa_0^I R_1^I \kappa_1^I R_2^I \kappa_2^I R_3^I \ldots \kappa_{q-1}^I \kappa_q^I \] (19)
An additional bit \( \kappa_{-1} \) is required to represent \( +1 \) as \( \|\text{rnd}(d, q)\|_{\infty} \leq 1 \). We shall be handled as a special case. To reduce the number of address bits, the table can store corresponding values for \( \|\text{rnd}(d^R, q)\| \) and \( \|\text{rnd}(d^I, q)\| \)
\[ \|\text{rnd}(d^R, q)\| = 0.\alpha_2^R \alpha_3^R \ldots \alpha_q^R \] (20)
\[ \|\text{rnd}(d^I, q)\| = 0.\alpha_2^I \alpha_3^I \ldots \alpha_q^I \] (21)
which eliminates the need for bits \( \kappa_0^R \) and \( \kappa_0^I \) (the sign) to be used when forming an address. Likewise, since \( \|d\|_{\infty} \geq \frac{1}{2} \)
we know that either \( \alpha_1^R = 1 \) or \( \alpha_1^I = 1 \) [6] (or both). Had an address been formed using
\[ \alpha_1^R \alpha_2^R \alpha_3^R \ldots \alpha_q^R \kappa_1 \alpha_2^I \alpha_3^I \ldots \alpha_q^I \]
then the address would require 2q bits. Given that
\[ \gamma(d^R, d^I) = \frac{1}{\|d^R + id^I\|^2} = \frac{d^R - id^I}{(d^R)^2 + (d^I)^2} \] (22)
\[ \gamma(d^I, d^R) = -\gamma(d^R, d^I) \] (23)
we could check if \( \alpha_1^R = 1 \), if so then the address is formed via
\[ \alpha_2^R \alpha_3^R \ldots \alpha_q^R \alpha_1^I \alpha_2^I \alpha_3^I \ldots \alpha_q^I \]
otherwise, it must be true that \( \alpha_1^I = 1 \) so the address is formed as
\[ \alpha_2^I \alpha_3^I \ldots \alpha_q^I \alpha_1^R \alpha_2^R \alpha_3^R \ldots \alpha_q^R \]
and the results obtained from the table look-up are negated based on (23). This reduces the number of address bits to \( 2q - 1 \) (halving the memory required) while introducing little additional overhead.

Extra care must be taken with the aforementioned approach; although it is true that dividend is assumed to be bounded by \(-1 < \|d\|_{\infty} < 1 \), it is certainly not true that \(-1 < \text{rnd}(d^R, q) < 1 \) in fact \(-1 \leq \text{rnd}(d^R, q) \leq 1 \) (same holds for \( \text{rnd}(d^I, q) \)). The two’s complement representation of the rounded divisor shown in equations (18) and (19) has range \([-1, 1] \). Negating -1 in two’s complement with the given representation is a special case; recalling that +1 is also a special case, the input is divided into two cases: \( \|\text{rnd}(d, q)\|_{\infty} < 1 \) and \( \|\text{rnd}(d, q)\|_{\infty} = \pm 1 \).

Another special case occurs when negating the results obtained from the table look-up due to the swapping discussed earlier. For positive values of \( d^R \) and \( d^I \) the real part of \( 1/d \) is positive and the imaginary part negative. The real part of \( 1/d \) is positive for positive values of \( d^R \) and the imaginary part of \( 1/d \) is negative for positive values of \( d^I \). Since \( 1/2 \leq \|\text{rnd}(d, q)\|_{\infty} \leq 1 \) and the table only stores values for positive \( d^R \) and \( d^I \) values, then \( 0 < K^R \leq 2 \) and \(-2 \leq K^I \leq 0 \). Therefore the table should only contain the magnitude of the value, which can be represented in \( 2 + t \) bits—this will present no anomalies if 3 integer bits are used for the negated values, i.e., the ROM will store \( 2 + t \) bits, but the negated value will be \( 3 + t \) bits.

Here we describe the operation of the table incorporating the special cases,
\[ \text{rnd}(d^R, q) = \kappa_{-1} \kappa_0 \kappa_1 \kappa_2 \kappa_3 \ldots \kappa_q \]
\[ \text{rnd}(d^I, q) = \kappa_{-1}^I \kappa_0^I \kappa_1^I \kappa_2^I \kappa_3^I \ldots \kappa_q^I \]
\[ A^R = \|\text{rnd}(d^R, q)\| = \alpha_0 R_1 \alpha_1 R_2 \alpha_2 R_3 \ldots \alpha_{q-1} R_q \]
\[ A^I = \|\text{rnd}(d^I, q)\| = \alpha_0^I \alpha_1^I \alpha_2^I \ldots \alpha_q^I \]
\[ A = \begin{cases} \alpha_0 R_1 \alpha_1 R_2 \alpha_2 R_3 \ldots \alpha_q R_q & \text{if } \alpha_0 = 1, \\
(\alpha_0^I \alpha_1^I \ldots \alpha_q^I), \ldots \alpha_q^I, & \text{otherwise} \end{cases} \]
\[ A_s = \begin{cases} \alpha_0 R_1 \alpha_1 R_2 \alpha_2 R_3 \ldots \alpha_q R_q & \text{if } A^R = \pm 1, \\
(\alpha_0^I \alpha_1^I \ldots \alpha_q^I), & \text{otherwise} \end{cases} \]
\[ (U^R, U^I) = \begin{cases} (1/2, -1/2) & \text{if } A^R = 1, A^I = 1, \\
(1/2, 1/2) & \text{if } A^R = 1, A^I = -1, \\
(-1/2, -1/2) & \text{if } A^R = -1, A^I = -1, \\
(-1/2, 1/2) & \text{if } A^R = -1, A^I = -1, \\
\text{REM}_c[A_s] & \text{if } A^R = \pm 1 \text{ and } A^I \neq \pm 1 \\
\text{or } A^I = \pm 1 \text{ and } A^R \neq \pm 1 & \text{otherwise} \end{cases} \]

\[ \begin{align*}
neg^R & = \begin{cases} 1 & \text{if real and imaginary swapped,} \\
0 & \text{otherwise} \end{cases} \\
neg^I & = \begin{cases} 1 & \text{if real and imaginary not swapped,} \\
0 & \text{otherwise} \end{cases} \\
K^R & = (1)\neg^R U^R \\
K^I & = (1)\neg^I U^I \end{align*} \]

From Table I, we have \( q = 6 \) and \( t = 6 \) for radix \( r = 4 \) and \( a = 3 \).
be computed via,
\[
x = (K^R + iK^I)(z^R + iz^I) \\
= (K^R z^R - K^I z^I) + i(K^I z^R + K^R z^I)
\]
\[
y = (K^R + iK^I)(d^R + id^I) \\
= (K^R d^R - K^I d^I) + i(K^I d^R + K^R d^I)
\]

Since multipliers are costly in hardware, the complex valued products will be computed one at a time. Coincidentally, \( y = Kd \) is not required until after the residuals have been initialized with \( x = Kz \), which can be computed in the previous cycles. Figure 4 shows the block diagram for the scaling module. The module uses several signals to control the data path: \( e_{in} \), \( e_{pres} \), \( e_{sc} \), and \( sel_{mul} \). Control signals \( e_{in} \) are clock enable signals to registers to control when data is latched. Clock enables on registers are used to facilitate multi-cycle paths which are necessary due to the larger delay of the prescaling logic.

In Fig. 4 \( e_{in} \) controls when the inputs to the complex division unit are latched such that the values can be retained throughout the course of the operation–this is not necessarily unique and depends on how the module is interfaced to other logic. For example, if the external logic feeds the arguments to the complex division unit in two cycles: sending \((z^R, z^I)\) in the first and \((d^R, d^I)\) in the second, then only 2 register banks are required for the inputs as opposed to 4. The current design reflects the assumption that the module receives its arguments in the same cycle, i.e., \((z^R, z^I, d^R, d^I)\).

Signal \( e_{pres} \) controls storing of the results of the prescaling ROM look-up, retained throughout the course of the operation. Signals \( sel_{mul} \) and \( e_{sc} \) are used to share the multipliers so that prescaling of the dividend and the divisor occurs in separate prescaling cycles. Although the prescaled value \( x = Kz \) is also fed through the registers controlled by \( e_{sc} \), its value is not retained but over-written in the next cycle by \( y = Kd \). The same enable signal \( (e_{sc}) \) is used once more to assure that the value of \( y \) is retained in these registers which feed the recurrence modules discussed in Section II-C.

### B. Bounds of Values

It is important to characterize the bounds of the inputs to the complex division module in addition to the bounds of the prescaled values which predetermine the width of inputs to the recurrence modules.

The input \( d \) is in the range \( \frac{1}{2} \leq \|d\|_{\infty} < 1 \), and through our convergence analysis further constrained \( \|Kd - 1\|_{\infty} < \epsilon_s \). This implies that the prescaled value \( x = Kz \) satisfies

\[
\max(|y^R|,|y^I|) < \epsilon_s \\
\Rightarrow |y^R| < \epsilon_s \quad |y^I| < \epsilon_s
\]

Since \( |y^R| < 1 + \epsilon_s \), its representation in two’s complement would require 2 integer bits and \( n \) fractional bits.
Likewise, the constraint (14) determines the maximum value that the residual could possibly take. For our design point \( \sigma = 4 \) which means that the residual is bounded by,

\[
\|w[j]\|_\infty = \frac{1}{4} \left( 3 + \frac{1}{2} + 2^{-4} \right) = 57/64
\]
\[\Rightarrow \|w^R[0]\| = |x^R| \leq 57/64
\]
\[\|w^I[0]\| = |x^I| \leq 57/64
\]

Therefore, the prescaled value \((x^R, x^I)\) requires only a single integer bit, and \(n\) fractional bits. We are interested in determining a bound on \(z\) which we can derive from the bound on \(w\),

\[
\|w[0]\|_\infty = \|Kz\|_\infty \leq 2\|K\|_\infty \|z\|_\infty \leq 57/64
\]

(24)
since \(\|K\|_\infty \leq 2\) then

\[
\|z\|_\infty \leq 57/256
\]

(25)
requiring only \(n - 1\) fractional bits, with most significant bit having weight \(2^{-2}\).

C. Digit-Recurrence Iterations

The digit-recurrence iterations compute the residuals (5) (6) and perform quotient-digit selection based on a short non-redundant estimate of the residuals as shown in Eq. (10) and (11).

The recurrences in (5) and (6) are structurally the same. Namely,

\[
w[j + 1] = 4w[j] + \sigma_1 y^R + \sigma_2 y^I
\]

(26)
The residuals are computed in redundant form in order to reduce the cycle time by eliminating the need for long carry chains. In our implementation we used a carry-save form. The operation is expressed as

\[
(w_C[j + 1], w_S[j + 1]) = ADD_{[6:2]}(4w_C[j], 4w_S[j], \sigma_1 y^R, 2\sigma_2 y^R, \sigma_2 y^I, 2\sigma_2 y^I)
\]

(27)
where \(ADD_{[6:2]}(a, b, c, d, e, f)\) is a \([6 : 2]\) carry-save adder taking 6 inputs and producing a carry vector and sum vector, shown in Fig. 5. The digits \(\sigma_1\) and \(\sigma_2\) are in the digit set \(-3, \ldots, 3\) so we implement this digit multiplication by decomposing \(\sigma_k = 2\sigma_k^2 + \sigma_k^1\) where \(\sigma_k^1 \in \{-1, 0, 1\}\). Multiplying by negative one is achieved by inverting the input and adding a carry-in to the reduction module. A block diagram of the structure used to compute the real recurrence is shown in Fig. 6.

Digit selection is performed by taking a short precision estimate of the residual and rounding it to the nearest integer via a small CPA and table. In the discussion that follows we generally say residual without referring specifically to the real or imaginary part—the analysis holds for both residuals \(w^R\) and \(w^I\). In Section II-B we determined that the residual has a single integer bit and \(n\) fractional bits, i.e., it is of the form \(w = w_i, w_1 w_2 \ldots w_n\) with
value $\sum_{i=0}^{n} w_i 2^{-i}$. In redundant form $w = w_c + w_s$ where $(w_c, w_s) = (C_0 C_1 C_2 \ldots C_n, S_0 S_1 S_2 S_3 \ldots S_n)$. Recalling that selection is performed via,

$$ q_{j+1} = \text{Sel}(\text{est}(4w[j], \sigma)) $$

where $\sigma = 4$ as determined in section II-A, we know that

$$ \text{est}(4w[j], 4) = w_0 w_1 w_2 w_3 w_4 w_5 w_6 = \sum_{i=0}^{n} w_i 2^{-i+2} $$

$$ \text{est}_{ERR}(4w[j], 4) < 2^{-4} $$

now since $w$ is in redundant form,

$$ \text{est}(4w_c[j], 5) = C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 $$

$$ \text{est}(4w_s[j], 5) = S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 $$

$$ g = \text{est}(4w_c[j], 5) + \text{est}(4w_s[j], 5) $$

$$ \text{est}_{ERR}(4w_c[j], 5) + \text{est}_{ERR}(4w_s[j], 5) < 2^{-4} $$

which gives us the short precision estimate of the residual $g$. It is important to realize that $g \neq \text{est}(4w[j], 4)$ in general but that they commit the same maximum error $2^{-4}$ in their approximation of $w[j]$. The addition in equation (28) requires the CPA that we have been referring to during this discussion.

$$ g_{2g} - 190 91 \ldots 95 = $$

$$ CPA(C_0 C_1 C_2 C_3 \ldots C_7, S_0 S_1 S_2 S_3 \ldots S_7) \quad (29) $$

To round $g$ and take the integer part one can use a small table as in table II by introducing an additional variable $g_2 = g_2 + g_3 + g_4 + g_5$ (i.e. the logical or of bits $g_2$ through $g_5$). This table is a function of 5 bits and produces three bits of output (for the encoding of $q_{j+1}$) and will efficiently map to LUTs.

### D. Optimizing the Recurrence Implementation

A straightforward implementation of the recurrence is shown in Fig. 7. There are several opportunities for its optimization:

- Since the residual is in the range $(-57/64, 57/64)$, there is only one integer bit required to store the value of the residual. Based on this observation there is no need to find the sum of bits with weight greater than $2^0 = 1$.

- The recurrence implementation can be optimized in the most significant bits by using the non-redundant value computed for selection in the adder instead of the redundant form stored in the registers. Although addition of a short CPA delay to the most significant bits seems counter-intuitive to optimization, it turns out that for all fitting attempts to the Stratix II architecture this path was not the critical path—paths with routing delays dominated the critical path (short carry chains don’t exhibit routing delays as there are dedicated carry paths in Adaptive Logic Modules [15]). Since using the non-redundant portion didn’t introduce a new critical path and reduced the input bits it served as a pragmatic optimization technique. The non-redundant approximation $g$ computed for selection can be used in the addition as opposed to using the $[6 : 2]$ adders. This simplifies the $[6 : 2]$ adder
to a [5 : 2] adder requiring 4 lateral carries (as opposed to a conventional [5 : 2] adder which only requires 3 lateral carries) which we denoted as [5 : 2]^4—the lateral carries come from the previous [6 : 2] adder. The interface between the [6 : 2] adder, the [5 : 2]^4 adder and the XOR slice is shown in Fig. 9.

- The [5 : 2]^4 adder produces both \( s_{j+1}^{(2)} \) and \( c_{0j+1}^{(2)} \), it is unnecessary to produce \( s_{0j+1}^{(3)} \) with the same module since we will discard \( c_{2j+1}^{(2)} \). Bit \( s_{0j+1}^{(3)} \) is just the sum modulus 2 of all bits of weight 1 plus the lateral carries, which can be computed via exclusive-ors (XOR).

Applying all mentioned optimizations we get an improved design shown in Fig. 8.

### III. DESIGN METHODOLOGY AND RESULTS

#### A. Methodology

The proposed designs were written at the RTL level using VHDL and simulated for functional correctness with Modelsim-Altera Edition 8.1. They were mapped to an Altera Stratix II architecture using Quartus II 8.1 flow tools. The Quartus Classic Timing Analyzer was used to determine the timing characteristics of the circuit in addition to placing constraints on ROM look-up and prescaling registers to inform the tool of multi-cycle paths.

The multiplies performed in the prescaling module map to the Altera DSP blocks for precisions up to 36 bits—these modules support up to \( 36 \times 36 \) multiplication. It does not really make sense to go beyond this precision as the current design choice is targeted for an architecture which supports fast multipliers. For larger precisions it seems more sensible to design an efficient custom rectangular multiplier.

#### B. Implementation Area and Delay Characteristics

The results show the number of ALUTs (Adaptive LUTs [15]), of which there are two in every ALM (Adaptive Logic Module) : the basic building blocks for logic in Altera Stratix II devices. The DSP blocks on Stratix II architectures support either eight \( 9 \times 9 \) multiplies, four \( 18 \times 18 \) multiplies or one \( 36 \times 36 \) multiply. The proposed design is limited to the availability of multiplication units and therefore we have only reported results for two design points, one utilizing a single DSP block with four \( 18 \times 18 \) multipliers and the other using four DSP blocks each performing \( 36 \times 36 \) multiplies. The results include on-the-fly conversion costs.

The most common scenario we foresee a designer will face when determining the usefulness of a complex division unit is when comparing performance to a software based solution. One such software solution presented in [14] is based on the following,

\[
\frac{a + jb}{c + jd} = \begin{cases} 
\frac{a + b(d/c)}{c + d(a/c)} + j \frac{b - a(d/c)}{c + d(a/c)} & \text{if } |c| \geq |d| \\
\frac{a + b(d/c)}{c + d(a/c)} + j \frac{a - b(c/d)}{d + c(c/d)} & \text{if } |d| \geq |c|
\end{cases}
\]

which requires significantly more arithmetic operations, 4 conventional divisions ad 3 multiplications. A complex divider has been described in [16] implementing Smith’s formula with a pipelined multiplier, divider, and adder for an 8-bit precision (+4 guard bits). The scheme uses small number
of Xilinx Virtex-II slices and operates at 100 MHz. Another design for a complex divider is proposed in [17]. It uses an algorithm similar to the SRT division. It also has an efficient implementation and a latency for 15-bit precision of about 600ns, and a throughput of 1.6MHz. These two approaches are not comparable to our higher-radix approach in terms of speed. They have an advantage that there is no prescaling implementation and a latency for 15-bit precision of about 600ns.

**IV. Conclusions and Future Work**

We presented the design and implementation of a radix-4 complex division unit with a single prescaling table. The implementation on an Altera Stratix II FPGA device requires 1185 ALUTs, with a critical path of 5.764 ns, and a maximum frequency of 173.49 MHz. The prescaling table requires 2K words of 16 bits. To our knowledge no comparable implementation exists at the time and our results initiate a point of reference for other hardware based designs. In future work we plan on exploring the use of multipartite tables to reduce the table requirements in addition to developing specialized rectangular multipliers to enable higher radix designs.

**Acknowledgments.** We thank Altera Corporation for providing the tools and FPGA devices used in this research.

### TABLE III

<table>
<thead>
<tr>
<th>Precision [bits]</th>
<th>16</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUTs</td>
<td>566</td>
<td>1185</td>
</tr>
<tr>
<td>DSP Block (9-bit elements)</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>Registers (FFs)</td>
<td>318</td>
<td>598</td>
</tr>
<tr>
<td>M4K RAM blocks</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Critical path (ns)</td>
<td>5.685</td>
<td>5.764</td>
</tr>
<tr>
<td>Max. frequency (MHz)</td>
<td>175.90</td>
<td>173.49</td>
</tr>
<tr>
<td>Prescaling look-up (Cycles)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Prescaling (Cycles)</td>
<td>2*4</td>
<td>2*4</td>
</tr>
<tr>
<td>Total prescaling (Cycles)</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Iterations (cycles)</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Total time (ns)</td>
<td>108</td>
<td>156</td>
</tr>
</tbody>
</table>

**REFERENCES**