A Novel Design of a Two Operand Normalization Circuit

Elisardo Antelo, Montserrat Bóo, Javier D. Bruguera, and Emilio L. Zapata

Abstract—This paper presents a new design for two operand normalization. The two operand normalization operation involves the normalization of at least one of two operands by left shifting both by the same amount. Our design performs the computation of the shift by making an OR of the bits of both operands in a tree network, encoding the position of the first nonzero bit. The encoded position is obtained most significant bit first, and then there is an overlapping with the shifting operation. The design we propose replaces two leading zero detector circuits and a comparator, that are present in the conventional approach. Our scheme demonstrates to be more area efficient than the conventional one. The circuit we propose is useful in floating point complex multiplication and COordinate Rotation Digital Computer (CORDIC) processors.

Index Terms—Digital VLSI design, floating point operations, leading zero detector circuit, normalization.

I. INTRODUCTION

There are many applications where a two operand normalization step is needed [3], [5], [7]. This implies the shift of the two operands until one of them is normalized in a certain range (i.e., [1, 2) for IEEE floating point arithmetic). Typical applications that require this operation are floating point operations over complex numbers like multiplication [7] or addition/subtraction, fixed-point CORDIC (COordinate Rotation Digital Computer) arithmetic for angle calculation in SVD (singular value decomposition algorithm) [4], [5], and the normalization of one operand expressed in carry-save redundant arithmetic, useful in arithmetic coders [1] and to detect the sign of a carry-save operand.

Conventional two operand normalization requires leading zero detection, comparison and shifting. In this paper, we present an area efficient design for the two operand normalization problem based on the encoding of the position of the first nonzero bit from the most significant side of both operands by means of a tree network of OR operations. As the position of the first nonzero bit is obtained most significant bit first, there is an overlapping with the shift of the operands. Our scheme avoids the comparison of the number of leading zeros of both operands, resulting in a more area efficient implementation maintaining the speed. The structure of the paper is as follows. Section II reviews the conventional two operand normalization. In Section III, we propose the new scheme. In Section IV, we compare both the conventional and the new design. Section V presents two applications, and finally Section VI reports some conclusions.

II. CONVENTIONAL TWO OPERAND NORMALIZATION

Two operand normalization implies to left shift both operands in such a way that at least one of them has a one (or a zero if the numbers are negative two’s complement) in the most significant bit position. For a low complexity implementation the left shift can be done in several cycles making a single left shift in each cycle, and then, it is only necessary to check the most significant bit of both operands in each cycle. This scheme has a very low hardware complexity but requires \( O(n) \) cycles for normalization (where \( n \) is the wordlength of the operands) and there is a variable number of cycles for normalization (data dependent processing) leading to a more complex control of the whole system. In this paper we consider a more efficient architecture that produces the result in one cycle. Fig. 1 shows a fast architecture for the two operand normalization, in one cycle, of two floating point operands with 24 bit significants and 8 bits for the common exponent (this scheme is used, for example, in floating point multiplication of complex numbers [7]). Although this is a particular implementation, it is easy to generalize it to any number of bits.

Two leading zero detector (LZD) circuits [6] encode the position of the first non zero bit from the most significant side in each operand. Then, by means of a comparator the smallest of the encoded values is determined. The position of the first non zero bit is exactly the shift that has to be performed to normalize at least one of the operands. The encoded position has a binary expression \( S = d_1d_2d_3d_4d_5d_6d_7d_8d_9d_10 \). This
Fig. 2. New algorithm for two operand normalization.

number is used to left shift both operands and it is also subtracted from the exponent. The binary value of $S$ has $\log_2(p)$ bits in a general case, where $p$ is defined as $p = 2^{\frac{n}{2} \log_2(n)}$ for $n$ bit operands. Observe that $p$ is the largest integer power of two greater or equal to $n$, and therefore $p$ is always even.

The left shift is usually performed by means of a barrel shifter to normalize the operands in one cycle [9]. The barrel shifter that we consider in this paper has to allow hardwired left shifts by $p/2$, $p/4$, $\cdots$, 1 in different stages (multiplexers). In Fig. 1, we have $n = 24$ and $p = 32$. After the normalization process, a rounding of the operands must be carried out for floating point arithmetic. Moreover, hardware for overflow/underflow detection in floating point arithmetic is needed (not shown in the figure).

As shown in Fig. 1, if the significants are expressed in two’s complement, a module is necessary to perform the one’s complement of the operands when the sign bit is one, to encode the position of the first non zero bit of both operands by performing an OR operation over the two operands in a tree network. The encoded position is produced msb (most significant bit) first, so the shifting operation begins when the most significant bit of the encoded position is available, and then there is an overlapping between the shift computation and the shifting operation. The algorithm we present has a binary tree structure where a new $d_i$ (beginning with the msb of $S$) is obtained in each decision of the tree. Fig. 2 shows the binary tree that represents our algorithm for the two operand normalization. In this figure we present the algorithm for 16 bit operands, but the structure of the algorithm can be extended easily to any number of bits (not necessary a power of two). Note that in the case of a non power of two wordlength of the operands, the tree would not be symmetrical. We also show in this figure, an example for the normalization of two operands. Note the path followed in the tree for this case to find the suitable shift. For each decision in the tree we show the result of the OR operation over certain bits, and the suitable value for $d_i$.

The encoding of the position of the first non zero bit is as follows. For each operand, we calculate the position of the first non zero bit and then we perform an OR operation between the two operands. The resulting position is then used as the shift value for the operands.

III. A NEW ALGORITHM FOR THE TWO OPERAND NORMALIZATION

In this section we describe the new algorithm for two operand normalization. We present an scheme that replaces the two LZD circuits and the comparator present in the conventional architecture. For the sake of simplicity we develop the algorithm for the two operand normalization of integer unsigned numbers, but it can be generalized for two’s complement signed integers, and floating point numbers, just replacing the two LZD circuits and the comparator in Fig. 1 by the new scheme.

The basic operation of our algorithm is to encode the position of the first non zero bit from the most significant side of both operands by performing an OR operation over the two operands in a tree network. The encoded position is produced msb (most significant bit) first, so the shifting operation begins when the most significant bit of the encoded position is available, and then there is an overlapping between the shift computation and the shifting operation. The algorithm we present has a binary tree structure where a new $d_i$ (beginning with the msb of $S$) is obtained in each decision of the tree. Fig. 2 shows the binary tree that represents our algorithm for the two operand normalization. In this figure we present the algorithm for 16 bit operands, but the structure of the algorithm can be extended easily to any number of bits (not necessary a power of two). Note that in the case of a non power of two wordlength of the operands, the tree would not be symmetrical. We also show in this figure, an example for the normalization of two operands. Note the path followed in the tree for this case to find the suitable shift. For each decision in the tree we show the result of the OR operation over certain bits, and the suitable value for $d_i$.

The encoding of the position of the first non zero bit is as follows. First we check the $p/2$ (OR operation over 8 bits for $p = 16$) more significant bits of both operands. If all the bits are zero, then the position of the first non zero bit in one of the operands is greater than $p/2$ and then $d_3 = 1$ (that is a left shift by $2^{n/2}$ have to be done). Else $d_3 = 0$ since the position of the first non zero bit is within the $p/2$ most significant bits. To determine the value of $d_2$, $p/4$ bits of each operand have to be checked. The position of these $p/4$ bits is a function of $d_3$. If $d_3 = 0$, then the $p/4$ msb have to be inspected. Else, the $p/4$ bits following the $p/2$ msb have to be checked. The remaining bits of the encoded position are determined following a similar procedure until $d_0$ is obtained. In the example shown in Fig. 2 we have drawn an square over the bits inspected in each case.

The algorithm we propose can be mapped efficiently in a combinatorial network. The normalization architecture based on the new algorithm is depicted in Fig. 3 for $n = 16$ ($p = 16$). We distinguish two parts, one corresponding to the shift calculation and another to the
performed a VLSI implementation in 0.7
conventional design described in section II. We have made the
we implement the OR of all possible groups of bits shown in Fig. 2,
Fig. 3. Hardware implementation of the new algorithm.
levels. For these comparisons we have considered the two operand
normalization of unsigned integer numbers with $n = 32$.
We now give the components of the critical path of both designs.

In this section we briefly discuss two interesting applications for
floating point multiplication of complex numbers. To increase the dynamic range of

| Table I
<p>| Comparison between Conventional and New Scheme |</p>
<table>
<thead>
<tr>
<th>Design</th>
<th>Conventional</th>
<th>New Design</th>
<th>Ratio (conv./new design)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay (ns.)</td>
<td>9.5</td>
<td>9.0</td>
<td>1.06</td>
</tr>
<tr>
<td>Total area ($mm^2$)</td>
<td>1.24</td>
<td>1.05</td>
<td>1.17</td>
</tr>
<tr>
<td>Area for shift computation ($mm^2$)</td>
<td>0.24</td>
<td>0.08</td>
<td>2.91</td>
</tr>
<tr>
<td>Gate levels</td>
<td>41</td>
<td>40</td>
<td>1.04</td>
</tr>
<tr>
<td>Gate count (shift computation)</td>
<td>302</td>
<td>107</td>
<td>2.8</td>
</tr>
</tbody>
</table>


IV. COMPARISON

In this section we perform a comparison of our design and the
conventional design described in section II. We have made the
comparison based on two different approaches. On one hand we have
performed a VLSI implementation in 0.7 µm CMOS double metal
technology of both schemes. On the other hand we have evaluated
the architectures based on a model independent of the technology. In
this model the hardware complexity is evaluated counting the gates
needed for the implementation and the delay is computed in gate

barrel shifters that perform the shift. For the calculation of the shift
we implement the OR of all possible groups of bits shown in Fig. 2,
and the correct path in the tree is selected by means of multiplexers
controlled by the value obtained for $d_i$ in each step. To reduce the
hardware complexity of the algorithm, bit $d_0$ is computed by checking
the most significant bit of both operands after the $\log_2(p) - 1$ initial
shifts in the barrel. For a general case, the circuit is made up of a
tree of OR gates with $\log_2(p)$ levels. There are $\log_2(p) - 2$ levels
of multiplexers.

V. APPLICATIONS

In this section we briefly discuss two interesting applications for
the scheme we propose, although many others are possible.
1) Floating Point Multiplication of Complex Numbers: In digital
signal processing (DSP), many signals are efficiently repre-
sented as complex numbers. To increase the dynamic range of
these signals, a floating point representation is used, leading to floating point operations in digital signal processors. In [7], a floating point multiplier for complex numbers is presented. The floating point representation of the complex numbers consist in two significant with their own sign bits, and one common exponent for both real and imaginary parts. This representation proves to be efficient from the point of view of memory requirements, and the processing is more efficient since only one normalization and rounding process is necessary at the end. After the multiplication both the real and imaginary parts of the result can be denormalized, due to cancellation in the additions, and a normalization and rounding process is needed. Since the real and imaginary parts have a common exponent, the numbers are assumed to be normalized to the greater of two, so a two operand normalization is necessary. The scheme we propose in this work can be used efficiently in this structure reducing the hardware requirements for normalization, maintaining the speed of the circuit. Since the complex number multiplier is a key hardware element in DSP, the proposed circuit for two operand normalization can be widely used in this field. Our scheme for normalization can also be used in floating point addition/subtraction of complex numbers.

2) Angle Calculation with CORDIC: This is an iterative algorithm to perform plane rotations and to evaluate trigonometric functions [4], [5]. The basic iteration is based on shift and addition operations. Many CORDIC-based algorithms have been proposed for signal processing, image processing, matrix algebra and robotics [4]. In [5], it has been shown that if the input vector is not normalized, then there can be large errors in the computation of the inverse tangent function (vectoring mode). This situation occurs in the evaluation of the SVD of a matrix based on fixed-point CORDIC arithmetic [5]. For the computation of the SVD, fixed-point format can be used without loss of accuracy. This way the overhead due to floating-point processing is avoided. However as the processing is in fixed-point format, the processors that evaluate the inverse tangent function have to incorporate, internally, a two operand normalization circuit to avoid errors.

VI. CONCLUSIONS

We have presented a novel design for two operand normalization. The new architecture is based on the computation of the shift from the most significant bit to the least significant one. This technique allows significant savings in area because it is not necessary to detect the number of leading zeros of the two operands and then to perform a comparison as in the conventional design. The main advantage of our architecture is the low cost in area as compared to the conventional one but maintaining time performance. The proposed circuit can be used efficiently in floating point operations over complex numbers and CORDIC processors, although other applications are possible.

ACKNOWLEDGMENT

The authors thank the referees for their valuable comments.

REFERENCES