Abstract

The design of efficient structures for multiplication of complex numbers is a critical aspect of many advanced architectures, in particular for applications requiring massive computation or high throughput. A new approach is presented in this paper for complex numbers in full fractional two's complement representation. A class of multipliers is discussed and evaluated: we consider in particular the computational time, the throughput and the silicon area required by a VLSI implementation. High regularity and modularity are some of the most interesting features of our architecture.

1 Introduction

Digital multipliers have been intensively studied for many years in the literature of our multiplier. The goal of such research is the identification of fast techniques for the execution of the basic operations involved in multiplication. The classical scheme for multiplication is based upon two operations: first, the generation of the partial products of the multiplicand and each bit of the multiplicator, then the addition of partial products. A number of architectures has been studied in the past: parallel input multipliers, serial input multipliers and parallel-serial input schemes have been proposed in the literature. The common goal has been the generation of the final result of the product with the smallest logical delay [1,2,5,6].

In particular, considerable interest has been given to serial-input multipliers, since they are highly suitable for VLSI implementation of large dedicated circuits for massive computation, e.g. in signal and image processing. In fact, even though serial-input serial-output multipliers are often apparently slower than parallel schemes, they have some appealing characteristics: they require only two input pins and a single output pin, so that only a reduced number of simple interconnections is necessary, their internal structure is simplified, they occupy a smaller area on the silicon substrate and they result more easily testable.

Several multipliers with a zero input latency have been proposed in literature [3,4]. However, in some cases, a non-null latency can be accepted if a smaller complexity can be achieved in the implementation. For instance, a serial input real multiplier with (n-2) latency, if n is the bit length of the product factors, has been proposed in [5] and modified in [6] to obtain a pipelined operation, in order to output products of 2n bits every n+1 clock cycles. Recently, a new serial-input multiplier with a pipelined architecture has been proposed [1,2]. It has a very simple hardware scheme despite of a slightly higher latency (n+1). It can reach a clock period limited only by the propagation delay of a single full adder and requires no delays in the presentation of the operands.

The present paper proposes a new approach in designing complex multipliers, which is partially based upon Dadda's scheme [7]. In our multiplier, the real and imaginary parts of the operands are represented in full fractional, two's complement notation, and they are presented to the inputs in bit-serial form. Our architecture optimizes the hardware structure by compacting the real multiplications into which a complex multiplication is decomposed. Overlapping and compression of some of these operations allow to reduce the computational time and the silicon area required by a VLSI implementation.

The basic computational remarks, upon which we build the multiplier architecture, are described in section 2. In section 3 we present the complete structure of our multiplier. It is partitioned into bit-slices that are stacked upon one another to constitute a semistochastic one-dimensional array with pipelined characteristics.

The addition part of the bit-slice is analyzed to define the most convenient structure for serial adders, in terms of silicon area used by the circuits and frequency of operation. Different architectural alternatives for adder structures are proposed in section 4. Depending on the type of adder the resulting one-dimensional array is characterized by an unilinear or by a bilinear topology.

2 Computational Remarks

The multiplication of two complex numbers can be reduced to four real multiplications and two additions/subtractions. Assume that the operands are \( z = a + ib \) and \( y = c + id \) and that the product \( z \) is:

\[
z = p + iq = x \times y = (a + ib)(c + id) = (ac - bd) + i(ad + bc)\]

where \( p \) and \( q \) are real numbers. In a more general definition, floating-point real numbers should be considered.

The corresponding multipliers, i.e. the ones that map directly the definition (1) of the complex multiplication, have a large size and require a long computational time. In a large number of applications, that require massive computation, input data are acquired from the external environment by using suited input modules and analog-to-digital converters. Typical examples are digital signal processing for robotics, image processing, aerospace applications and system control. In most of these cases, input and output data are represented in full fractional two's complement fixed point notation.

More efficient multipliers can be designed under these assumptions. For simplicity, we assume that the real and imaginary parts of both operands are \( n \) bits long. Our reasoning and the multiplier architecture could be easily adjusted to consider input data with different length, i.e. different number of bits. The real and imaginary parts of the result should be represented by means of \( 2n + 1 \) bits, since each of them is the result of two real multiplications of \( n \) bits factors and one addition (or one subtraction) of \( 2n \) bits operands, i.e. the results of the addition/subtraction. However, without any loss of generality, we can assume that both the real and imaginary parts of the result are represented over \( 2n \) bits. In fact, if the actual range of values of the \( n \) bits input data can generate results which are \( 2n + 1 \) bits long, we can adjust \( n \) to reflect the previous constraint on the output range. The new number \( N \) of bits we consider is given by \( N = n + 1 \): under such assumption, it is possible to prove that the result is represented by \( 2N \) bits, without any need of truncation. The user can choose also a different approach to satisfy the previous constraint: instead of increasing the number of bits of input data, it is possible to reduce the range of input data so that the result is represented over \( 2N \) bits, by truncation, rounding or augmentation.

The smallest size for the VLSI implementation of arithmetic units can be achieved by adopting a bit-serial architecture. Moreover, some researchers [10] have shown that it is possible to reach very high computational performances also with a bit-serial computation. In particular, when large arrays of processing units must be considered for massive computation, it is often more convenient to use a bit-serial VLSI (or WSI) architecture, instead of a parallel approach, which is apparently more efficient. The small size of bit-serial processors allows to integrate on the same component (chip or wafer) a higher number of serial processors than of the functionally equivalent parallel ones. Data transfer between bit-serial processors may become faster, since interconnections are placed on the same component, instead of running between adjacent components on computing boards. This performance increase may completely overcome the reduction due to the serialization of data transfer and manipulation.

In our multiplier, we consider therefore bit-serial input operands and results. The real and imaginary parts of the operands are supplied to the multiplier simultaneously on separate input lines and the real and imaginary parts of the result are output serially from two separate output lines. Both inputs and outputs are represented in full fractional two's complement LSB-first arithmetic; the multiplier implements the traditional LSB-first multiplication algorithm. However, it is possible
to show that a similar architecture can also be designed for the MSB-first arithmetic.

From the literature [3,5], it is known that the simplest and more natural way to compute the multiplication is the serial/parallel algorithm. Therefore, in our multiplier architecture we adopt this approach. In this algorithm one of the operands is presented in parallel to the multiplier unit, while the second one is inserted serially. The partial product can be easily computed for each bit of the bit-serial operand by ANDing such bit with the other operand. The final product is obtained by accumulating the partial results at each clock period. This algorithm is well suited and natural if the bits of one of the operands are presented in parallel to the multiplier. Since we consider two bit-serial inputs, a serial-to-parallel conversion is needed inside the multiplier. We will see in the next section that such operation is practically costless for our architecture.

The presence of a serial-to-parallel conversion leads to consider also the possibility of extending the use of the architecture to the serial/parallel scheme for data presentation: one input is supplied serially, while the other is presented in parallel to the multiplier. Such scheme for data presentation avoids obviously the necessity of a serial-to-parallel converter. It can be used, for instance, if the parallel operand is fixed or if it is infrequently changed.

3 Architecture of the multiplier

The multiplication of complex numbers could be computed by using four real multipliers working in parallel; their results are added (or subtracted) by two adders/subtractors to generate the real and imaginary parts of the final complex number. This computational structure wastes a large silicon area, since each arithmetic operation is performed separately from the other ones.

The computational structure of the multiplier, as pointed out in the previous section, leads to notice that homogeneous operations can be effectively performed in a single step. The basic idea to design our multiplier is that the sums of the partial product rows of the partial product matrices contributing to the imaginary (real) part of the result can be performed together with the final addition (subtraction) of two real products to generate the imaginary (real) part of the complex result. By squeezing all these additions (subtractions) into a single, though more complex, operation, we achieve a very high performance, thanks to the high pipelining of the computation. Besides, the silicon area required by such approach is smaller, since we can compact the addition section and the shift registers for serial-to-parallel conversions.

The overall architecture of the multiplier is presented in fig.

As soon as the multiplicand $a + ib$ has been completely introduced into the multiplier, it is possible to present serially the bits of the multiplicator $c + id$. For every new bit of the multiplicator, four product rows of the partial product matrices that are associated to the four real multiplications $ac$, $bd$, $ad$ and $bc$, respectively, are generated. Each partial product row is obtained by multiplying all the bits of the operands $a$, $b$, $c$ and $d$ of the multiplicator and with the current two bits of the parts $c$ and $d$. Four rows of AND logic gates are used in the generator of partial products to compute the four product rows, producing the four combinations of two parallel data with two serial data. At each computational cycle $i$, i.e. at each clock period, the architecture generates a new group of four rows, one for each matrix of the partial products for the four real multiplications $ac$, $bd$, $ad$ and $bc$. Such rows are $(ac)_i$, $(bd)_i$, $(ad)_i$, and $(bc)_i$, respectively, where $1 \leq i \leq n$.

The real and imaginary parts of the result are successively computed by properly adding the groups of four rows. In particular, as stated by the mathematical definition of the complex multiplication, we must subtract the partial product row $(bd)_i$ from the partial product row $(ac)_i$ to obtain a partial sum of the real part of the result and we must subtract $(ad)_i$ to $(bc)_i$ for a partial term of the imaginary part. Such operations are performed in the two addition (or subtraction) sections for the multiplier, the former is dedicated to the real part of the result, the latter to the imaginary part.
Each addition section is composed by a \( n \) bits adder/accumulator and by two or three shift registers. The adder/accumulator contains a chain of full adders and two or three rows of flip-flops for each bit of the multiplicand. These flip-flops hold the carries generated by the full adders, so that it is possible to reduce the computation time by avoiding an immediate propagation of carries through the accumulated partial sum. These adder/accumulator structures are generalizations of the well known carry-free adder schemes. In other words, the partial sum is represented by two or three rows of bits, depending on the specific solution adopted. There are three different solutions, which are discussed in section 4. Carry propagation is carried out at each clock period by properly shifting the contents of such flip-flops.

When the multiplicator \( c+id \) has been completely introduced into the multiplier, only the \( n \) least significant bits of the real and imaginary parts are generated and output by the adders/accumulators. The data to compute the most significant bits are still stored in the flip-flops of the adder/accumulator. This part of the result can be generated by completing the carry propagation in the shift registers. The contents of the flip-flops of the adder/accumulator are transferred in parallel into the shift registers. These shift registers are shifted and added by a serial adder connected to their outputs. This final adder on the right hand side of the shift registers in each addition section performs this operation and generates the \( n \) most significant bits of the real and imaginary parts, starting from the least significant bit.

In fig. 2 we give the complete time diagram of data and result flows in our multiplier architecture. It is clearly shown that the multiplicand \( a+ib \) must be loaded before the multiplicator \( c+id \). The loading of \( a \) and \( b \) is performed serially from two different input lines; \( n \) clock periods are required to complete this data transfer. During the

\[
\begin{align*}
1: & \quad a_i \quad a_{i+1} \quad a_{i+2} \quad a_{i+3} \\
2: & \quad b_1 \quad b_{i+2} \quad b_{i+3} \\
3: & \quad c_1 \quad c_{i+1} \quad c_{i+2} \\
4: & \quad d_{i+1} \quad d_{i+2} \\
\end{align*}
\]

...and the four real multiplications \( ac, bd, ad \) and \( be \) are performed. At the same time these partial products are added together according to the algorithm described above.

In the same \( n \) clock periods, the least significant \( n \) bits of the real and imaginary parts of the \( (i+1) \)th product, \( p \) and \( q \), are delivered serially from the two output lines.

In the next \( n \) clock periods, the addition sections go on generating the most significant parts of the \( (i+1) \)th product. These \( n \) bits are emitted in \( n \) clock periods; at the same time, the multiplicand \( a+ib \) of the \( (i+2) \)th product is loaded into the shift registers. No superposition occurs with the current contents of these shift registers (which are the most significant halves of the parts of the \( (i+1) \)th product), as the incoming bits of \( a \) and \( b \) flow in the same direction of the contents of the shift registers.

Overlapping of output delivering and input acquisition is required to exploit the hardware capabilities of the structure fully, without introducing idle clock periods and wasting computation time. In fact, a complete overlapping leads to use adders and shift registers continuously and, thus, to the maximum throughput allowed by the specific hardware technology.

In the previous description of the data flow of the multiplier, we have assumed that the operand \( a+ib \) precedes \( c+id \). This choice is strictly required by our multiplier: the partial product rows can be computed, four at a time, as soon as two bits of the bit-serial multiplicator \( c+id \)

\[
\begin{align*}
\text{parallel register cell} & \quad \text{parallel register cell} \\
& \quad \text{cell adder acc.} \\
& \quad \text{shift registers cell} \\
\end{align*}
\]

...are supplied, only if this constraint is satisfied. Consider the four real numbers \( a, b, c, d \) and the four real multiplications \( ac, bd, ad \) and \( be \). The numbers \( a \) and \( b \), which are supplied during \( n \) clock periods, cannot contribute to the products being currently output, due to the nature of the algorithm itself. Due this constraint, we need supply \( c+id \) before \( c+id \), or vice versa.

The high regularity of the computation leads directly to introduce a highly modular hardware structure for our multiplier. The multiplier is realized as a semisystolic array of bit-slices. The architecture of the bit-slice is shown in fig. 4. Each bit-slice computes one bit for each one of the four partial product rows, one bit for each one of the two accumulated partial sums and one bit for each one of the shift registers. By connecting \( n \) bit-slices, we can build the multiplier for \( n \) bits operands. The only irregular components of the architecture are the two serial adders at the end of the shift registers in the addition sections.

The flexibility of our architecture is a basic feature to design new multipliers in a short time for a given application. Moreover, it provides the capability of designing functionally reconfigurable multipliers. In fact, we can place \( S \geq 1 \) bit-slices that can be cascaded statically, for instance, by using fuses, or dynamically, for instance, by means of programmable switches. The VLSI or WSI implementation can contain a number of generic multipliers, which can be customized to the specific application by choosing the length of the operands and hence the number of bit-slices.

4 Addition section

In the multiplier architecture presented in the previous section, the sum (or subtraction) and accumulation of partial products can be performed by using different structures of adders for the addition sections.

As already said, each addition section is dedicated to the generation of the real and imaginary parts of the complex product. The basic function of each addition section is the accumulation of the current partial product row to the partial sum produced during the previous clock period. Therefore, in the adder unit of each bit-slice, at least four inputs are needed:

- The bit of the partial product row of one real term of the complex multiplication, e.g. \( ac \).
The adder unit of the bit-slices computes:

- The bit of the partial product row of another real term of the complex multiplication, e.g. $b_d$.
- The bit of the result coming from the adjacent bit-slice.
- The carry bits generated during the previous clock periods. These carry bits may vary in number and weight, depending on the specific carry-free scheme adopted.

The adder unit of the bit-slices computes:

- One bit output to the adjacent bit-slice, in the direction of the final multiplier output, i.e. to the least significant position.
- A number of carries with different weights, depending on the solution adopted.

These carries can be stored in the addition section or transferred to the adjacent ones.

Alternative solutions can be designed to implement the addition section according to the functional definition discussed above. In fig. 5 we present three possible structures: the first one is based upon parallel counters (fig. 5.a), the second one uses a cascade of two full adders connected in a feed-forward layout (fig. 5.b), while the third one adopts a cascade of two full adders in which some carries are propagated backward along the adder itself (fig. 5.c). The simplest structure considers a parallel counter (fig. 5.a). This parallel counter computes the classical three-out-of-six output representation of the inputs: the least significant bit of the result is delivered to the adjacent bit-slice on the right hand side, while the two carries are latched in the same bit-slice in specific flip-flops for the subsequent clock period. Note that the inputs are only five, but the input d has a double weight. This kind of structure for the addition section allows to connect the bit-slices in a one-dimensional unilateral array, in which data and result flow in a single direction, from left to right.

Reduction of the complexity of the circuit can be achieved by using traditional full adders connected in cascade, as shown in fig. 5. In this structure, the partial products and the result coming from the adjacent bit-slice are added to generate a preliminary sum $s_1$ and a preliminary carry $c_1$. The preliminary sum is then added to the carries which have been generated during the previous clock periods and held in suit flip-flops. The preliminary carry is used to update these flip-flops for the subsequent clock periods. Only one output bit is delivered to the adjacent bit-slice towards the least significant side of the adder/accumulator. As the structure of fig. 5.a, also this solution uses a one-dimensional unilateral array of bit-slices to implement the pipelined multiplier.

The third solution is shown in fig. 5.c. It is still based upon a cascade of traditional full adders, but flip-flops are drastically reduced by rearranging the interconnections between adjacent bit-slices. The difference between this solution and the previous ones is mainly in the interconnections that propagate backwards some carries to provide a proper synchronization of data transfer during the computation. Such links ensure in fact a one-stage delay, which is required by the algorithm, without introducing additional flip-flops. In this case, bit-slices can be connected to create a one-dimensional, bilateral array, i.e. an array in which data flow in two opposite directions. Note that backward propagation is strongly limited: during each clock period, the back-propagating signals are transferred within a distance of two bit-slices.

All the architectural solutions described above behave as serial adders/accumulators. During the first $n$ clock periods, the adder section produces the least significant $n$ bits of the result by accumulating the partial products. At the end the most significant part of the result resides accumulated inside the storage devices of the addition section. In these flip-flops the result is represented in redundant form. These bits are transferred into the shift registers below the adder section, as shown in fig. 3. During the subsequent $n$ clock periods, these bits are progressively shifted and added together to generate the final representation of the most significant part of the result. During the same clock periods, the new multiplicand $a \cdot i$ for the second following multiplication is acquired and stored into these shift registers. The real part $a$ is progressively stored into one shift register of the addition section which produces the real part of the result, while the imaginary part $b$ is stored into one shift register of the addition section for the imaginary part of the result.

The three architectural solutions, presented in fig. 5, have different costs in terms of silicon area and computational time. In solution (a), the parallel counter requires a relevant number of logic gates and a high number of storage devices, but it is the fastest structure, as it could be designed as a two-level combinatorial circuit. Solution (b) requires fewer logic gates than (a), despite of a higher computational time due to the cascade of two full adders. Solution (c) is the best compromise between silicon area and computation time since it has fewer flip-flops than (a) and (b) and few logic gates, even if there is a cascade of full adders.

In this section we have discussed the structure for the addition section assuming that all partial products are added together. This is strictly correct only for the generation of the imaginary part of the result, for which only two's complement additions are required. As we have said in section 2, the generation of the real part of the complex product requires the addition of one partial product row and the subtraction of the second partial product row to and from the accumulated partial sum. Subtraction can be easily converted into an addition by introducing a complementation stage at the output of the generator of partial products. Such stage consists of an array of NOT logic gates, one for each bit of the partial product row. This stage computes the one's complement of the partial product row; the two's complement representation can be obtained later by storing it in the adder/accumulator. Such adjustment can be performed at reasonable costs in terms of additional silicon area and computational time, without affecting the throughput of the multiplier. In the architectural solutions of fig. 5.a and 5.b, the final serial adder on the right hand side of the shift registers is implemented as a cascade of two serial full adders, as shown in fig. 5.a. In the solution 5.c, we can use a single serial full adder. Some results about the characteristics of

![Figure 5: Structures of adders.](image)

![Figure 6: Compression of the redundant form of the MSB's of the result in the addition sections: parallel counters and feed-forward full adders (a) - back-propagating full adders (b).](image)

the multiplier architectures discussed in this paper are summarized in tab. 1. HA are half adders, FA are full adders and FF are flip-flops. Tab. 1 only considers the adder section of the bit-slice (excluding the shift registers); the product generator is the same one for the three solutions (a), (b) and (c).

<table>
<thead>
<tr>
<th>Solution</th>
<th>$n^a$ HA</th>
<th>$n^b$ FA</th>
<th>$n^c$ FF</th>
<th>$n^d$ Logic Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>a Logic Gates = many</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Bit-slice characteristics.
A new approach for the design of multipliers for complex numbers has been proposed and evaluated in this paper. Serial multiplication has been considered, since it often allows to achieve the highest computational performances at the minimum cost in terms of silicon area for a VLSI implementation.

The main characteristic of our pipelined architecture is the capability of executing complex multiplication by using the silicon area of about two real multipliers, instead of the four real multipliers required by the traditional mathematical definition of this operation. In our architecture we obtained a full pipelining for operands acquisition and result generation and delivering. This performance allows to reach the maximum clock frequency which is compatible with the adopted VLSI implementation of serial structures.

Multiplier compactness can be exploited to design small and fast arithmetic units for dedicated applications requiring massive computation and high throughput, e.g. in digital signal and/or image processing for real-time systems. Modularity and regularity may be useful to implement adaptable units, which are capable of functional reconfiguration as for the length and the number of operands, according to the actual precision and the amount of computation required by the application.

As for testability is concerned, theoretical analyses have shown that optimized architectures are fully testable in linear time with respect to the number of bit-slices, under a single stuck-at fault model. Besides, it is possible to prove that no additional inputs and outputs are required to perform the test [11].

Concurrent error detection can be introduced into this architecture by using AN coding techniques [9]; the additional cost in terms of redundant silicon area is very small, with respect to the detection capability.

Hardware reconfiguration of the cascade of bit-slices can be obtained by using a "Diogenes" approach [7,8], in order to add fault-tolerance capabilities to the overall structure. In this case, after fault localization the faulty bit-slice can be switched off by adjusting properly the interconnection switches.

References