EFFICIENT FPGA-BASED REALIZATION OF COMPLEX SQUARER AND COMPLEX CONJUGATE USING EMBEDDED MULTIPLIERS

Shuli Gao, Dhamin Al-Khalili, Noureddine Chabini, and Pierre Langlois[+]

Department of Electrical and Computer Engineering, Royal Military College of Canada
Kingston Ontario, Canada
[+]Department of Computer Engineering, École Polytechnique de Montréal
Montréal Québec, Canada

ABSTRACT

This paper presents an efficient design methodology and a systematic approach for the implementation of squaring of complex numbers and their conjugate, using small-size embedded multipliers. Various benchmarks were tested for operands with size ranging from 19 to 85 bits targeting Xilinx Spartan-3 FPGA. Our proposed approach was compared with the traditional technique. The results illustrate that our design approach is very efficient in terms of timing and area saving. For the complex squarer, the combinational delay is reduced by an average of 16.8% and area saving, in terms of 4-inputs LUTs, is about 27.2%. For the complex conjugate realization, combinational delay and area are reduced by about 18.6% and 41.6%, respectively.

I. INTRODUCTION

The complex domain is widely used in real-time signal processing and image processing [2][3]. Some of the important functions required to realize various algorithms are the complex squarer, and conjugate functions [1] that represent the square of the magnitude.

The complex squarer and conjugate functions are expressed as given by the following equations:

$$(x \pm jy)\cdot(x \pm jy) = x^2 - y^2 \pm j(2\cdot x\cdot y) \quad (1)$$

and

$$(x + jy)\cdot(x - jy) = x^2 + y^2 \quad (2)$$

where x and y are unsigned integers. From these expressions, it is clear that one needs squarers and multipliers to realize these complex functions, besides adders/substracters to get the final result.

Squaring is a special case of the multiplication operation since it requires multiplying an operand by itself. Multiplication is a costly arithmetic operation compared to, for instance, an addition or a subtraction. Consequently, to efficiently realize the complex functions described by expressions (1) and (2), then a starting point is to efficiently realize the required multiplication and squaring operations.

Efficient realization of multipliers and squarers for integer numbers has been addressed extensively in the literature. In [4], a high-speed unsigned squarer was presented. The design approach in [4] replaces low order partial product bits with a ROM Lookup Table in order to reduce the delay of the final carry propagate adder. In [5], a fast parallel squarer based on divide-and-conquer strategy was proposed. For the approach in [5], the operand of the squarer is divided by a factor 2, 4, or 8. Then, 1-bit to 4-bit squarers and multipliers are used as building blocks to realize the large size squarer. By carefully designing these building blocks, the quality of the resulting squarer can be improved in terms of speed and area. In [6], a recursive design approach for using small multipliers to implement a large size multiplier was proposed. The large operands of the multiplier are recursively split until getting operands that can be multiplied using small-size multipliers. Only C-language based simulation results have been provided in [6] to assess the approach instead of carrying out real implementations.

Even though some clever algorithms targeting Application Specific Integrated Circuit (ASIC) designs are proposed for realizing multiplication/squaring operations, they would have a limited value or even no value when FPGA (Field Programmable Gate Array) is the targeted design platform.

FPGA vendors are now offering highly optimized embedded hardwired multipliers as one of resources available to designers. A typical example is Xilinx Spartan-3 family [9]. For instance, the device 3s5000fg1156 contains 104 on-chip 18-bit×18-bit multipliers. Consequently, for applications to be implemented on modern FPGAs, it seems completely natural to use these embedded multipliers to carry out a multiplication or squaring operation instead of designing a multiplier or a squarer circuit from scratch.

While designers can now use on-chip n-bit×n-bit multipliers to speedup the processing of multiplication-intensive and/or squaring-intensive applications, one question is how to realize multiplication and/or squaring operations when the size of operands exceeds n bits? A possible solution is to use a divide-and-conquer strategy [7][8], which is not new and has been already proposed in some approaches like those in [5][6][7]. In this strategy, the multiplication and/or squaring operations are broken-down into multiplications involving operands not exceeding n bits, followed by a set of addition operations. The multiplication and/or squaring operations can now be carried out efficiently using embedded multipliers. However, to reduce the latency and the area of the final design, one needs to solve the problem of decreasing the number of addition operations in this strategy and realizing them efficiently. This paper proposes a new approach for solving the problem.

This paper is organized as follows. Sections II and III present our proposed optimized design approach for carrying out large size multiplications and squaring functions using on-chip n-bit×n-bit embedded multipliers, which are required to realize (1) and (2) for large bit width x and y. Assessment of the proposed approach is the focus of Section IV. Conclusions are given in Section V.

II. A NEW DESIGN APPROACH FOR THE MULTIPLICATION OF LARGE INTEGERS

This section focuses on presenting our proposed design approach for the multiplication of large unsigned integers using n-bit×n-bit embedded multipliers. The binary representation of the multiplication is:

$$Z = X \cdot Y = [x_{k-1}x_{k-2}\ldots x_{n-1}x_{n-2}\ldots x_1x_0]_2 \cdot [y_{k-1}y_{k-2}\ldots y_{n-1}y_{n-2}\ldots y_1y_0]_2 \quad (3)$$

where X and Y are the inputs of the multiplier, k is the size of the inputs of the multiplier and n is the size of the inputs of the embedded multipliers.

Due to space limitation, instead of starting with examples, we will directly present our proposed approach for the case of operands with any size (i.e., for any value of k such that k > n). We will first present how to align the partial products, and then how to efficiently organize the required additions.

A. Aligning the Partial Products

As a general case, a multiplier with k-bit inputs should be split into m segments where k is such that $(n\times(m-1)) < k \le (n\times m)$. The input operands of the multiplier can then be expressed as follows:

$$X = [x_{k-1}\, x_{k-2} \ldots x_{n-1}\, x_{n-2} \ldots x_1\, x_0]_2 = [X_{m-1}\, X_{m-2} \ldots X_1\, X_0]_{2^n} = 2^{(m-1)n}X_{m-1}+2^{(m-2)n}X_{m-2}+\ldots+2^nX_1+X_0 \quad (4)$$

$$Y = [y_{k-1}\, y_{k-2} \ldots y_{n-1}\, y_{n-2} \ldots y_1\, y_0]_2 = [Y_{m-1}\, Y_{m-2} \ldots Y_1\, Y_0]_{2^n} = 2^{(m-1)n}Y_{m-1}+2^{(m-2)n}Y_{m-2}+\ldots+2^nY_1+Y_0 \quad (5)$$

where each of X_i and Y_i terms has n bits for $0 \le i < (m-1)$. For i = m-1, these elements have a width of (k-(m-1)n) bits. Using expressions (4) and (5), the output of the multiplier is expressed as follows:

$$\begin{aligned}
Z &= X \cdot Y \\
&= (2^{(m-1)n}X_{m-1}+2^{(m-2)n}X_{m-2}\ldots2^nX_1+X_0)\cdot(2^{(m-1)n}Y_{m-1} \\
&\quad +2^{(m-2)n}Y_{m-2}+\ldots+2^nY_1+Y_0) \\
&= 2^{(m-1)n}X_{m-1}(2^{(m-1)n}Y_{m-1}+2^{(m-2)n}Y_{m-2}+\ldots+2^nY_1+Y_0) \\
&\quad +2^{(m-2)n}X_{m-2}(2^{(m-1)n}Y_{m-1}+2^{(m-2)n}Y_{m-2}\ldots2^nY_1+Y_0) \\
&\quad \ldots \\
&\quad + 2^nX_1(2^{(m-1)n}Y_{m-1}+2^{(m-2)n}Y_{m-2}+\ldots+2^nY_1+Y_0) \\
&\quad + X_0(2^{(m-1)n}Y_{m-1}+2^{(m-2)n}Y_{m-2}+\ldots+2^nY_1+Y_0) \\
&= (2^{2n(m-1)}X_{m-1}\cdot Y_{m-1}+2^{2n(m-2)}X_{m-2}\cdot Y_{m-2}+\ldots+2^{2n}X_1\cdot Y_1 \\
&\quad +X_0\cdot Y_0)+(2^{2n(m-2)+n}X_{m-1}\cdot Y_{m-2}+2^{2n(m-3)+n}X_{m-2}\cdot Y_{m-3} \\
&\quad +\ldots+2^{3n}X_2\cdot Y_1+2^nX_1\cdot Y_0)+(2^{2n(m-2)+n}Y_{m-1}\cdot X_{m-2} \\
&\quad +2^{2n(m-3)+n}Y_{m-2}\cdot X_{m-3}\ldots+2^{3n}Y_2\cdot X_1+2^nY_1\cdot X_0) \\
&\quad +(2^{2n(m-2)}X_{m-1}\cdot Y_{m-3}+2^{2n(m-3)}X_{m-2}\cdot Y_{m-4}+\ldots \\
&\quad +2^{4n}X_3\cdot Y_1+2^{2n}X_2\cdot Y_0)+(2^{2n(m-2)}Y_{m-1}\cdot X_{m-3} \\
&\quad +2^{2n(m-3)}Y_{m-2}\cdot X_{m-4}+\ldots+2^{4n}Y_3\cdot X_1+2^{2n}Y_2\cdot X_0)\ldots \\
&\quad +(2^{nm}X_{m-1}\cdot Y_1+2^{n(m-2)}X_{m-2}\cdot Y_0)+(2^{nm}Y_{m-1}\cdot X_1 \\
&\quad +2^{n(m-2)}Y_{m-2}\cdot X_0)+(2^{n(m-1)}X_{m-1}\cdot Y_0)+(2^{n(m-1)}Y_{m-1}\cdot X_0)
\end{aligned} \quad (6)$$

According to the number of bits of X_i or Y_i (i = m-1, m-2, ...1, 0) stated above, the sizes of the partial products in expression (6) are given in Table 1.

The partial products in expression (6) can be organized as shown in Fig. 1, which are stacked in rows: Z_0 to Z_{2m-2}. All the partial products here can be executed using n-bit×n-bit embedded multipliers.

Table 1. Sizes, in bits, of partial products in expression (6).

Partial Products	Number of Bits
$X_{m-1} \cdot Y_{m-1}$	2×(k-(m-1)×n)
$X_i \cdot Y_j$, where i or j = m-2, m-3,…,1, 0	2×n
$X_{m-1} \cdot Y_i$ or $Y_{m-1} \cdot X_i$, where i = m-2,…,1, 0	(k-(m-1)×n)+n

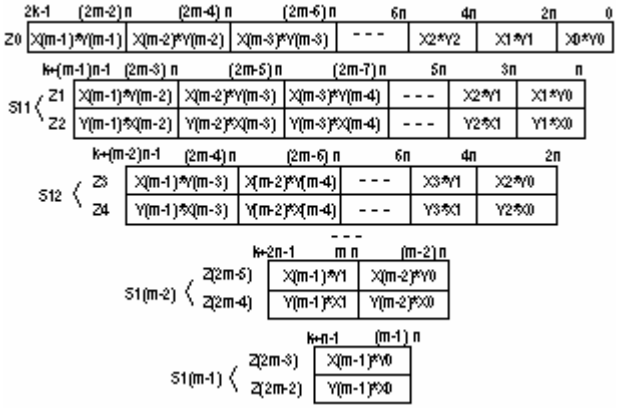


Figure 1: Organization of the partial products for an m-segment multiplier.

B. Organizing the Additions

For an m-segment multiplier, the first level of additions consists of adding each pair of Z_is together as shown in Fig. 1, where the adder $S_{1i} = Z_{2i-1} + Z_{2i}$, for i =1, 2…(m-1).

To get the output of the multiplier, other levels of additions are required. Adders for these additions have to be designed carefully in order to improve the performance and reduce the area of the design. The main approach to reduce the combinational delay is to incorporate parallelism and to achieve additions with small size operands. The latter will also impact area saving. Fig. 2 illustrates how the adders, at the second level, are organized. There are two cases to be considered: 1) m is an even number, and 2) m is an odd number. Fig. 2 (a) and (b) demonstrate these two cases separately.

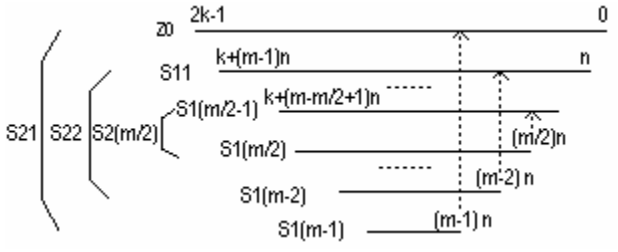
Fig. 2 (a) shows the organization of the addition operations when m is an even number. In this case, the addition operations at the second level are executed as follows:

$$\begin{aligned}
S_{21} &= Z_0 + S_{1(m-1)} \\
S_{22} &= S_{11} + S_{1(m-2)} \\
&\ldots \\
S_{2(m/2)} &= S_{1(m/2-1)} + S_{1(m/2)}
\end{aligned} \quad (7)$$

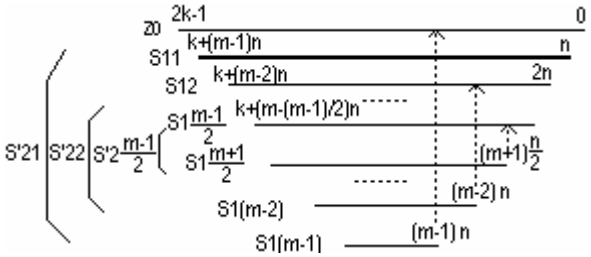
where S_{2v} (v = 1, 2, …, m/2), the "2" in the subscript designates the addition level number, and v is for labeling the addition operations being executed at this level.

In equations (7), the additions are grouped such that the adders to be used are with operands of width as small as possible. Also, for each addition, the least significant bits (i.e., the bits to the right of the dashed vertical lines shown in Fig. 2 (a)) are directly propagated to the output free from additions.

Since Z_0 has a width identical to the one of the output of the multiplier, then it is not necessary to reserve one bit



(a) m is an even number.



(b) m is an odd number.

Figure 2: Organization of additions for an m-segment multiplier.

carry for S_{21} when the addition is computed. The width in bits, denoted by W_{e0}, for the output of S_{21} is:

$$W_{e0} = (2k-1) - n \times (m-1) + 1 \quad (8)$$

Except for S_{21}, the outputs of all adders required to perform the sum operations in (7) have the same size, denoted by W_e. It can be calculated from any adders at this level except for S_{21}. For instance, the adder $S_{2\,(m/2)}$ has a width equal to:

$$W_e = (k+(m-m/2+1)n - (m/2)n+1) + 1 = k+n+2 \text{bits} \quad (9)$$

where $(k+(m-m/2+1)n)$ is the position of the most significant bit of $S_{1(m/2-1)}$, and $(m/2)n$ is the position of the least significant bit of $S_{1(m/2)}$. The last term "1" in (9) stands for the one bit carry for this adder. Equation (9) indicates that the size, in bits, depends on k and n when m is even. For instance, when m=4, then $W_{e0} = 2k-3n$ and $W_e = k+n+2$.

Fig. 2 (b) shows the organization of the addition operations when m is an odd number. The additions at the second level are performed as follows:

$$S'_{21} = Z_0 + S_{1\,(m-1)}$$
$$S'_{22} = S_{12} + S_{1\,(m-2)}$$
$$\ldots$$
$$S'_{2\,(m-1)/2} = S_{1\,(m-1)/2} + S_{1\,(m+1)/2} \quad (10)$$

Since the number of operands in Fig. 2 (b) is also an odd number, one of them cannot be used at this level. In general case, the largest operand has to be used at the last level addition. Otherwise, the largest adder will be employed over and over. However, in (10), Z_0 is added to $S_{1(m-1)}$ even though Z_0 is the largest operand while S_{11} is not used at this level. This decision is made for the following reasons. Firstly, the data of Z_0 is available at the first level addition. Secondly, since the operand $S_{1(m-1)}$ has the smallest size among the first level adders, then it can be executed faster than other adders. Thus, the second level of additions can start as soon as possible. Finally, S_{11} has the next largest size among the operands shown in Fig. 2 (b).

When m is an odd number, all the adders at the second level have the same size in bits, except for $S'_{21}=Z_0+S_{1(m-1)}$. From Fig. 2 (b), the number of bits for most adders, denoted by W_o, can be obtained from $S'_{2\,(m-1)/2} = S_{1(m-1)/2} + S_{1(m+1)/2}$ or any other adder at this level. Hence, the number, in bits, of each adder at the second level is:

$$W_o = [k+ n(m-(m-1)/2) - n(m+1)/2 +1] + 1 = (k+2)\text{bits} \quad (11)$$

where $[k+n(m-(m-1)/2)]$ is the position of the most significant bit of $S_{1(m-1)/2}$, and $n(m+1)/2$ is the position of the least significant bit of $S_{1(m+1)/2}$. Also, the last term "1" in (11) is the carry bit of the adder. From equation (11), it is clear that the value W_o at the second level only depends on k when m is odd.

For the addition $S'_{21} = Z_0 + S_{1\,(m-1)}$, the size in bits of the adder is exactly the same as when m is an even number. It can be calculated by equation (8).

Again, to build the adders as small as possible, all the bits that are located on the right-hand sides of the dashed lines in Fig. 2 (b) are propagated directly to the relative outputs free from additions. Using this scheme allows to save the area, and reduce the delay for the adders.

As a summary, to get the output of an m-segment multiplier, $\lceil \log_2 (m) + 1 \rceil$ levels of addition are required. For instance, for m = 2, two levels of additions are required. In addition, there are three features in this proposed approach: 1) the adders in use are with operands of size as small as possible; 2) the adders at the same level operate in parallel as much as possible; and 3) except for the first level addition, most adders at the same level have the same size, and they cast almost the same combinational delay. All these properties decrease the length of the critical path, reduce the number of devices required, and efficiently save area.

III. THE DESIGN APPROACH FOR SQUARING OPERATIONS

A squarer can be derived by using a multiplier with same input operands, since $Z = X^2 = (X \cdot X)$. Therefore, there are some commonalities between a squarer and a multiplier. The rules we have presented for the multiplier design can be followed in the squarer design. For instance, splitting large size input operand of the squarer into several segments and concatenating some of the least significant bits to the result of additions are similar to what we did for designing the multiplier. However, some special issues have to be considered for a squarer design. First, when summing partial products, any $(X_i \cdot X_j) + (X_j \cdot X_i)$ reduces to $2(X_i \cdot X_j)$ which saves one multiplication since multiply by 2 is just a shift, and also saves one level adders. Second, to reduce execution time, one needs to sum a term having a large size with a term having a small size. Due to space limitation, a detailed version of our approach is available in [11].

IV. EXPERIMENTAL RESULTS

The multiplication and squaring operations required in the complex squarer and conjugate functions are implemented based on the approaches presented in Sections II and III. The resulting designs are called here the proposed complex squarer and the propased

conjugate functions. These designs have been coded in VHDL and synthesized for Xilinx' Spartan-3 family [9], targeting the device 3s5000fg1156. The size of the embedded multipliers is considered as 17 and the number of segments m is varied from 2 to 5. The obtained results are compared with the following two traditional approaches. The first traditional approach is coding the real part of the complex squarer as $(x^2 - y^2)$ and the imaginary part as $(2 \cdot x \cdot y)$, and is named as Direct Model in the rest of this paper. The second traditional approach is using Xilinx' complex multiplier IP-Core Generator, and is called IP-Core Model in the rest of this paper. The range of the operands has been varied from 19 to 85 bits for the proposed approach and the Direct Model. This range is reduced to 19 to 63 bits for the IP-core Model since the IP-Core Generator has a maximum input range of 63 bits for the complex multiplier. All of these designs are synthesized using the Xilinx ISE 7.1 XST tool. To compare these three approaches, the following metrics extracted from the synthesis reports are used: 1) Maximum propagation delay, 2) the number of LUTs (Look Up Tables), and 3) the number of embedded 18-bit×18-bit signed multipliers.

The results of timing of the complex squarer and the conjugate are shown in Fig. 3. Number of utilized LUTs and embedded multipliers are presented in Figs. 4 and 5, respectively.

Comparing to the Direct Model, our proposed technique has resulted in an average delay reduction of 16.8% and 18.63% for the complex squarer and the complex conjugate function, respectively. The number of 4-input LUTs has been reduced by 27.2% and 41.59% for the complex squarer and the complex conjugate function, respectively. Moreover, the proposed approach has decreased the number of embedded multipliers used by an average of 22.2% for the case of complex squarer and 33.27% for the case of the complex conjugate function.

Notice that, in all the cases, the results of the Xilinx' IP-core implementation for the complex squarer and the complex conjugate function are inferior to both of the Direct Model and our proposed approach.

V. CONCLUSIONS

The focus of this paper is to provide designers with an optimized solution to realize the squaring of a complex number and/or its conjugate using small-size embedded multipliers when a dedicated complex multiplier is not available. Our approach in this paper is based on divide-and-conquer strategy with a set of efficient schemes for realizing the required additions.

The proposed approach proves to be very efficient for the case of FPGA-based designs as it was demonstrated by the presented numerical results for the case of implementations targeting Xilinx FPGAs. The proposed approach has outperformed by far the solutions produced by the Xilinx ISE XST synthesis tool as well as the solutions derived using the IP-core generator.

REFERENCES

[1] Y. B. Mahdy, S. A. Ali, and K. M. Shaaban, "Algorithm and two efficient implementations for complex multiplier", *Proceedings of IEEE Conference on Electronics, Circuits and Systems*, vol.2, pp. 949-952, 1999.

[2] V. G. Oklobdzija, D. Villeger and T. Soulas, "Considerations for design of a complex multiplier," *Proc. of IEEE Conf. on Signals, Systems and Computers*, vol.1, pp. 366-370, 1992.

[3] V. Tarokh, H. Jafarkhani, and A. R. Calderbank, "Space-time block codes from orthogonal designs," *IEEE Transactions on Information Theory*, vol. 45, No. 5, July 1999.

[4] E. G. Walters, J. Schlessman, and M. J. Schulte, "Combined unsigned and two's complement hybrid squarers," *Asilomar Conference on Signals, Systems and Computers*, November 2001.

[5] J. T. Yoo, K. F. Smith, and G.Gopalakrishnan, "A fast parallel squarer based on divide-and-conquer," *IEEE J. of Solid-State Circuits*, vol. 32, No. 6, pp. 909-912, 1997.

[6] Albert N. Danysh and Earl. E. Swartzlander, "A recursive fast multiplier" *Asilomar Conference on Signal, Systems & Computers*, vol. 1, pp.197-201, 1998.

[7] N.Nedjah, and L. de Macedo Mourelle, "A reconfigurable recursive and efficient hardware for Karatsuba-Ofman's multiplication algorithm," *Proc. of IEEE Conference on Control Applications*, V.2, pp. 1076-81, 2003.

[8] Behrooz Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, New York, Oxford, 2000.

[9] Xilinx Inc. [On line]: http://www.xilinx.com/.

[10] S. Gao, N. Chabini, D. Al-Khalili, and P. Langlois, "Efficient realization of large integer multipliers and squarers," *Proc. of Tthe 4th International IEEE-NEWCAS Conference*, pp. 237-240, June 2006.

[11] Shuli Gao, Noureddine Chabini, Dhamin Al-Khalili, and Pierre Langlois, "An Optimized Design Approach for Squaring Large Integers Using Embedded Hardwired Multipliers," *Proc. Of The 4th ACS/IEEE International Conference*, pp. 248-254, March, 2006.

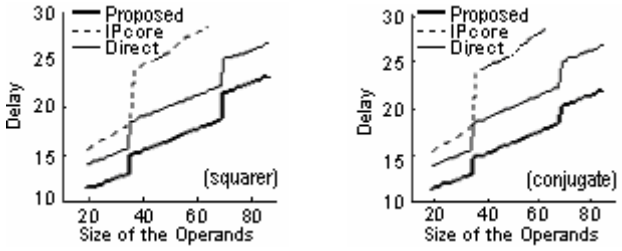


Figure 3: Maximum propagation delay.

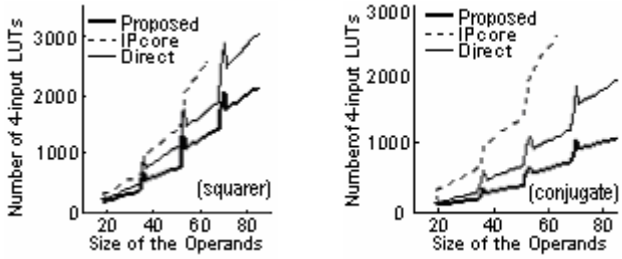


Figure 4: Number of 4-input LUTs used.

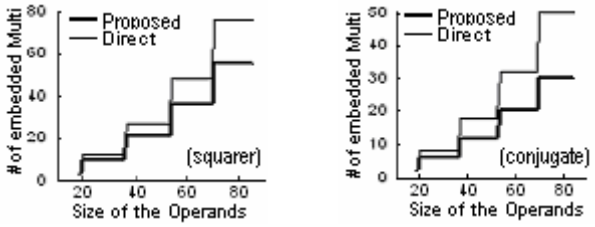


Figure 5: Number of embedded multipliers used.