Resource Optimal Squarers for FPGAs

Andreas Böttcher, Martin Kumm, Florent de Dinechin

Resource Optimal Squarers for FPGAs

Andreas Böttcher*, Martin Kumm*, Florent de Dinechin†
*Fulda University of Applied Science, Fulda, Germany
†Univ Lyon, INSA Lyon, Inria, CITI, Lyon, France
andreas.boettcher@cs.hs-fulda.de, martin.kumm@cs.hs-fulda.de, florent.de-dinechin@insa-lyon.fr

***Abstract*—Squaring is an essential operation in computer arithmetic that can be considered as a special case of multiplication where several simplifications can be applied to reduce the complexity of the resulting circuit. However, the design of a squarer is not straightforward for modern FPGAs that provide embedded DSP blocks and look-up-tables (LUTs). This work proposes a flexible method to design resource optimal squarers, i.e., a squarer that uses a minimum number of LUTs for a user-defined number of DSP blocks. The method uses an integer linear programming (ILP) formulation based on a generalization of multiplier tiling. It is shown that the proposed squarer design method significantly improves the LUT utilization for a given number of DSPs over previous methods, while maintaining a similar critical path delay and latency.**

***Index Terms*—squarer, multiplier tiling, compressor tree, integer linear programming, computer arithmetic**

I. Introduction

Squaring is an essential operation in computer arithmetic, frequently used in applications like function approximation using polynomials [1] or in the calculation of the Euclidean distance [2]. It is also useful in the implementation of other arithmetic components, for instance the square root: the Newton-Raphson and multiplicative methods [3, Section 7.3] as well as the Taylor series expansion used in the VFloat library [4, Fig. 6] all involve a square operation. Another square is needed to determine the correctly rounded value using any of the previous methods [5, Section II].

Squaring can be considered as a special case of multiplication, so many design principles from multiplier design can also be applied. But squarers contain some redundancies which can be exploited to reduce the complexity of the operation.

Like multiplication, the squaring operation can be described as the weighted sum of the partial products of each combination of the individual bits of the input vector according to

$$\begin{aligned} X^2 &= \left(\sum_{i=0}^{w_X-1} x_i 2^i\right) \cdot \left(\sum_{j=0}^{w_X-1} x_j 2^j\right) \\ &= \sum_{i=0}^{w_X-1} \sum_{j=0}^{w_X-1} x_i x_j 2^{i+j} \ . \end{aligned} \tag{1}$$

Using a radix of two, each multiplication forms one partial product bit and can be realized as a logical AND-operation of two bits. These have to be added in a bit-shifted way to compute the squarer result. This is usually performed by compressor trees [3].

Fig. 1 shows at the top the partial product matrix of a $w_X = 8$ bit radix-2 squarer, that we use as running example

$$\begin{array}{cccccccccccccccc}
2^{15} & 2^{14} & 2^{13} & 2^{12} & 2^{11} & 2^{10} & 2^{9} & 2^{8} & 2^{7} & 2^{6} & 2^{5} & 2^{4} & 2^{3} & 2^{2} & 2^{1} & 2^{0} \\
\hline
&&&&&&&& x_0x_7 & x_0x_6 & x_0x_5 & x_0x_4 & x_0x_3 & x_0x_2 & x_0x_1 & x_0x_0 \\
&&&&&&& x_1x_7 & x_1x_6 & x_1x_5 & x_1x_4 & x_1x_3 & x_1x_2 & x_1x_1 & x_1x_0 & \\
&&&&&& x_2x_7 & x_2x_6 & x_2x_5 & x_2x_4 & x_2x_3 & x_2x_2 & x_2x_1 & x_2x_0 && \\
&&&&& x_3x_7 & x_3x_6 & x_3x_5 & x_3x_4 & x_3x_3 & x_3x_2 & x_3x_1 & x_3x_0 &&& \\
&&&& x_4x_7 & x_4x_6 & x_4x_5 & x_4x_4 & x_4x_3 & x_4x_2 & x_4x_1 & x_4x_0 &&&& \\
&&& x_5x_7 & x_5x_6 & x_5x_5 & x_5x_4 & x_5x_3 & x_5x_2 & x_5x_1 & x_5x_0 &&&&& \\
&& x_6x_7 & x_6x_6 & x_6x_5 & x_6x_4 & x_6x_3 & x_6x_2 & x_6x_1 & x_6x_0 &&&&&& \\
& x_7x_7 & x_7x_6 & x_7x_5 & x_7x_4 & x_7x_3 & x_7x_2 & x_7x_1 & x_7x_0 &&&&&&& \\
&&&&&&& \Downarrow &&&&&&&& \\
&&&&&&& x_4 &&&&&&&& \\
&&&&& x_5 && x_3x_4 && x_3 &&&&&& \\
&&& x_6 && x_4x_5 & x_3x_5 & x_2x_5 & x_2x_4 & x_2x_3 && x_2 &&&& \\
& x_7 && x_5x_6 & x_4x_6 & x_3x_6 & x_2x_6 & x_1x_6 & x_1x_5 & x_1x_4 & x_1x_3 & x_1x_2 && x_1 && \\
& x_6x_7 & x_5x_7 & x_4x_7 & x_3x_7 & x_2x_7 & x_1x_7 & x_0x_7 & x_0x_6 & x_0x_5 & x_0x_4 & x_0x_3 & x_0x_2 & x_0x_1 && x_0 \\
\hline
\end{array}$$

Fig. 1: Partial product matrix (top) and simplified matrix (bottom) of an 8 bit radix-2 squarer.

throughout the paper. Each partial product combination $x_i x_j$ of the individual bits of the input vector X is calculated and inserted in the corresponding column of the resulting matrix according to its weight 2^{i+j}. Several well-known simplifications can be applied to the bit matrix of the squarer [3]. First, the terms on the diagonal can be transformed as

$$x_i x_i = x_i \ , \tag{2}$$

so no AND-gate is required to calculate those partial products. Second, as $x_i x_j = x_j x_i$, many partial products appear twice within one column, so the computation of

$$x_i x_j + x_j x_i = 2 x_i x_j \tag{3}$$

can be applied offline by just considering the term $x_i x_j$ in the subsequent column. Applying these two rules reduce about half the number of partial product bits as illustrated in Fig. 1 (bottom). Note that additional transformations can by applied to reduce the column height of the bit matrix and thus the stage count of the subsequent compressor tree [3]. However, we do not go into details here as the transformations are less relevant for FPGAs. The resulting architecture of the 8 bit squarer example is shown in Fig. 2. There the individual partial product bits from the modified AND-array are connected to the proper columns of the compressor tree (labeled as Σ).

The design of the necessary compressor trees on FPGAs has been well explored [6]–[12]. However, the generation of partial products in the radix-2 case using 2-input AND gates maps poorly to the look-up-tables (LUTs) of FPGAs. Furthermore, practically all modern FPGAs from vendors like Intel or Xilinx (AMD) feature embedded multipliers or DSP units, which should also be utilized.

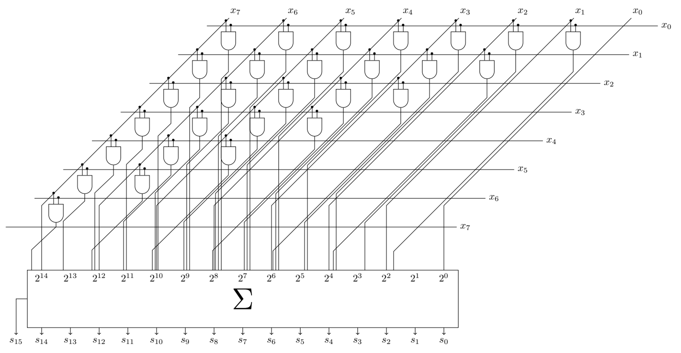

x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
Σ
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Fig. 2: Resulting circuit of an 8 bit radix-2 squarer.

Various previous works cover the efficient realization of squarers on FPGAs by improving the utilization of the DSP units and handling the weighting between DSP and LUT resources [13]–[15]. In the work of Lee and Burgess [14], a combination of efficient $2\times k$ multipliers together with a single DSP block is used to reduce its complexity. The works in [13] and [15] address large squarer design by using modifications of the Karatsuba-Ofman algorithm to save DSPs.

This work proposes a squarer design methodology that treats the partial product generation for squaring as an optimization problem that is formulated using integer linear programming (ILP). It is based on the idea of *multiplier tiling* [16] which is a systematic methodology for the partial product generation using smaller sub-multipliers for which an efficient LUT implementation exists or embedded DSP blocks. The main contributions of this work are (1) exploiting of redundancies that appear in squaring within the multiplier tiling framework, (2) incorporating the squarer specific optimizations in the tiling model, and (3) considering special squarer tiles. We provide a closed form ILP model that can be optimally solved using generic ILP solvers and show that this model scales well to real-world problem sizes. With optimality, we mean that for a given set of sub-multipliers and squarers (the tiles) and their associated resource cost, we will find a solution with least resource cost. Of course, the solution can only be as good as the tiles. Improved tiles would translate to better solutions when added to our method.

In the following we will briefly introduce the state-of-the-art in multiplier tiling which is later extended to the proposed squarer design methodology.

II. Multiplier Tiling

A. Basic Idea

A large multiplier (or squarer) of size $2w \times 2w$ can be implemented with four smaller sub-multipliers $M_{1,\dots,4}$ of size $w \times w$ by splitting the inputs into sub-words of size w:

$$\begin{aligned} X \times Y &= (X_1 2^w + X_0)(Y_1 2^w + Y_0) \\ &= \underbrace{X_1 Y_1}_{M_4} 2^{2w} + (\underbrace{X_1 Y_0}_{M_3} + \underbrace{X_0 Y_1}_{M_2}) 2^w + \underbrace{X_0 Y_0}_{M_1} \ . \end{aligned} \tag{4}$$

This can be graphically represented as a board of size $2w \times 2w$ which is *tiled* by tiles of size $w \times w$. Fig. 3a shows an example

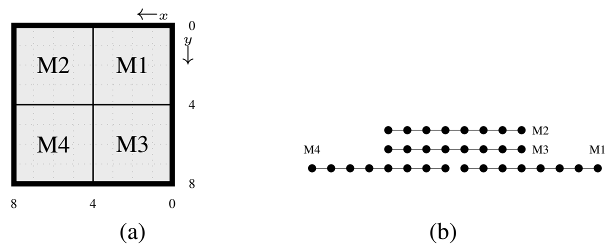


Fig. 3: Tiling (a) and dot diagram (b) of a 8×8-multiplier, composed of four 4×4 sub-multiplers M1 ...M4.

TABLE I: Properties of LUT- and DSP-based multipliers [19] (top) and squarer tiles (bottom) for Xilinx FPGAs

Shape	Tile area A_t	$\text{cost}_t^{\text{tile}}$	$\text{cost}_t^{\text{mult}}$	w_{out}	D^t	LUT Efficiency
1×1	1	1.65	1	1	0	0.61
1×2	2	2.3	1	2	0	0.87
2×3	6	6.25	3	5	0	0.96
3×3	9	8.9	5	6	0	1.01
2×k	$2k$	1.65k + 2.3	$k+1$	$k+2$	0	$\frac{2k}{1.65k+2.3}$
24×17	408	26.65	0	41	1	15.31
1×1	1	0.65	0	1	0	1.54
2×2	4	3.6	1	4	0	1.11
3×3	9	5.9	2	6	0	1.53
4×4	16	8.2	3	8	0	1.95
5×5	25	10.5	4	10	0	2.38
6×6	36	14.8	7	12	0	2.43

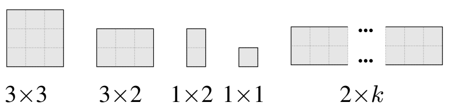


Fig. 4: Geometric shapes of the LUT-based tiles [19]

illustration of a 8×8 multiplier realized by four 4×4 sub-multipliers. With this tiling representation, multiplier design using heterogeneous computing resources such as DSP blocks or logic-based multipliers reduces to finding a *valid* tiling of the multiplier board. A *valid* tiling is given when every position on the board is covered by a tile without overlaps with other tiles. Overlaps at the border of the board are allowed as these result in an under-utilization of the sub-multiplier.. Once a valid tiling is found, the weight of the partial results can be directly determined by the position on the board according to the Manhattan distance relative to the origin (top right in Fig. 3a). Fig. 3b shows how the results of the multipliers have to be added. For example, the result of multiplier M4 has to be added in column $4 + 4 = 8$ as this is its Manhattan distance.

In an FPGA, different sub-multiplier options are available, providing different sizes at different LUT cost. Hence, the tiling problem is defined as finding a valid tiling with minimal cost for a given set of tiles (i.e., sub-multipliers). Various heuristic [17]–[19] and optimal [20], [21] methods have been proposed to solve the tiling problem and to trade between DSP and complementary LUT-resources.

B. Available Multipliers Tiles

The tiles are selected from a list of logic or DSP-based sub-multipliers shown in the upper part of Table I. The logic

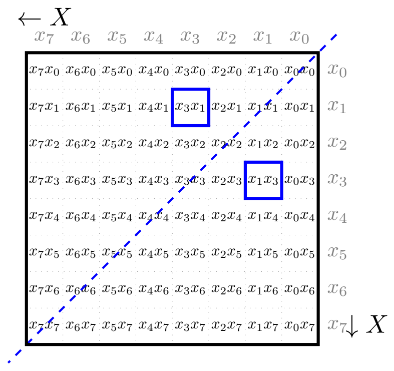


Fig. 5: Symmetry at the main diagonal

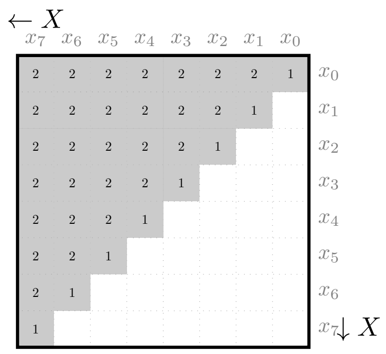


Fig. 6: Resulting weight of the partial products

multipliers are visualized in Fig. 4. The multiplier tiles are characterized by their shape and resulting tile area A_t they cover on the board. Note that the 1×1 squarer has zero cost as it corresponds to the input bit. Furthermore, each tile t has realization costs in terms of LUTs $\text{cost}_t^{\text{tile}}$ and might have a DSP count D^t. The LUT-cost is composed of the cost of the multiplier itself $\text{cost}_t^{\text{mult}}$, plus the cost to compress the w_{out} output bits generated by this tile in the compressor tree

$$\text{cost}_t^{\text{tile}} = \text{cost}_t^{\text{mult}} + \text{cost}^{\text{comp}} w_{\text{out}} \ . \tag{5}$$

The cost contribution of the compressor tree was experimentally estimated to be on average $\text{cost}^{\text{comp}} = 0.65\,\text{LUT/bit}$ for Xilinx Virtex 6, 7-Series and Ultrascale(+) target FPGAs [20]. A useful metric is the LUT-efficiency E_t of a particular tile, defined as

$$E_t = \frac{A_t}{\text{cost}_t^{\text{tile}}} \ , \tag{6}$$

turned out to be an important indicator for the evaluation of tiles [19]. According to this metric, the DSP has by far the highest efficiency, since it only entails LUT costs for the compression of the calculated partial product bits. Hence, DSPs are the preferred resource but they are also a limited resource and can be supplemented by additional LUT-based tiles to ensure an efficient implementation.

C. *Previous multiplier tiling ILP model*

The proposed ILP-based squarer design method is based on the multiplier tiling ILP model of [20]. The idea is that a solution is described by a set of decision variables $d_{x,y}^t$ which are true when tile t is placed at a certain position (x, y). One variable is present for each possible (x, y) position on the board and each possible tile.

The optimization problem is modeled as

$$\text{minimize} \sum_{t=0}^{T-1} \sum_{x=0}^{w_X-1} \sum_{y=0}^{w_Y-1} \text{cost}_t^{\text{tile}} d_{x,y}^t$$

subject to

$$\text{C1:} \sum_{t=0}^{T-1} \sum_{x'=0}^{x} \sum_{y'=0}^{y} o_{x-x',y-y'}^t \, d_{x',y'}^t = O_{x,y} \left. \begin{array}{l} \text{for } 0 \leq x < w_X, \\ 0 \leq y < w_Y \\ \text{with } O_{x,y} = 1 \end{array} \right\}$$

The objective is clearly to minimize the sum of costs of the individual tiles (as given in Table I) and a single set of constraints (C1) is used to allow only valid tilings. For that, $o_{x,y}^t$ is a set of binary constants that describes the shape of a sub-multiplier (true for places where it covers the board). $O_{x,y}$ is another set of binary constants that describes the shape of the overall multiplier. $O_{x,y}$ is usually set to one for all valid positions but can be set to zero for the least significant bits in truncated multipliers [16], [21].

III. PROPOSED OPTIMAL SQUARER DESIGN

A. *Squarer Design*

As discussed above, squarers can be considered as a special case of multipliers where both inputs are equal. We now look into the specialties when considering squaring in multiplier tiling. Fig. 5 shows the tiling representation of our 8 bit example squarer when represented as a 8×8 multiplier. It can be observed that, first, the partial products that can be simplified according to (2) are located at the diagonal (dashed blue line). Second, identical partial products that can be simplified according to (3) are symmetric along the diagonal (the example of x_3x_1 / x_1x_3 is highlighted by blue boxes).

To exploit this in the squarer design using tiling, this translates to the two following rules: First, partial products along the diagonal do not have to be covered (they are identical to the corresponding input bit). Second, only the partial products in the upper left (or lower right) triangle have to be considered when counting them twice in the compressor tree (by simply connecting them with the adjacent column with the next higher weight). The resulting relative weights of the partial product bits for the 8 bit radix-2 squarer from Fig. 5 is shown in Fig. 6.

The general, non-radix-2 case is a bit more complicated: Considering larger sub-multipliers than 1×1 may lead to situations where a sub-multipler may touch or cross the diagonal. All the different cases are illustrated in Fig. 7, which shows various interesting tile configurations:

- Tile M1 represents a usual 2×3 multiplier that does not overlap with the border. It computes the partial products of two symmetric areas shown by the black and gray boxes when taking it twice.

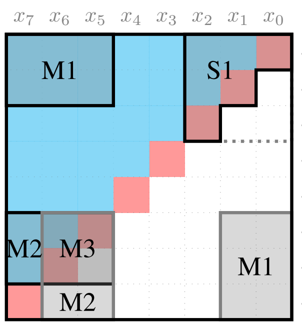

x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
M1
S1
M2
M3
M2
M1

Fig. 7: Examples of tiles in a squarer

- Tile M2 is also a usual 2×3 multiplier, but it is placed in such a way that it covers bits on the diagonal. A negative consequence is that a square of bits (noted M3) is computed twice, since M3 is covered both by M2 and by its mirror image. For a correct result, M3 must be recomputed and subtracted in the compressor tree. This may be globally beneficial if M2 (or a similar, larger tile) is computed by a DSP block that otherwise efficiently covers a large area.
- Tile S1 is a squarer-specific tile. It can only be placed on the diagonal for which the inputs are identical. Thus, S1 is only a 3-input function, covering a virtual 3×3 area that is larger than for the 2×3 multiplier which is a 5-input function. Hence, this is very beneficial for LUT-based squarers as discussed in the next subsection.

Fig. 9 already provides an example for a 53-bit squarer as required for double-precision floating-point. It can be observed that all the cases discussed above appear. Besides, the upper right corner is computed by a DSP block used as a squarer. This figure also motivates that this is a combinatorial optimization problem that is far from being trivial.

B. Logic Based Squarer Tiles

Modern Intel or Xilinx FPGAs feature 6-input LUTs (LUT6), what limits efficient multipliers to the size to 3×3 when using a plain tabulation of results. The LUT6 of the targeted Xilinx FPGA can realize two LUT5 with shared inputs. Thus, a 3×3 multiplier requires four LUT6 and two LUT5 (the two LSBs depend only on 4 inputs), so 5 LUT6 in total. Considering special squarer tiles allows simplifications as both input vectors are equal.

The lower part of Table I, shows the properties of these specialized squarer tiles and Fig. 8 shows their shape. It can be seen at that the squarer tiles have a considerably lower cost per area and reach more than twice the efficiency of the logic-based multiplier tiles in the upper part. Here, the 6-bit squarer performs best as bits 0 to 1 can be directly routed through, bits 2 to 5 and 9 to 10 depend on less then 5 inputs, requiring only seven LUT6 in total.

C. ILP Squarer Design Method

The main idea to exploit the ideas presented above for the squaring in the tiling is to extend the possible weight contributions of tiles on the board and to make use of the symmetry of the problem. First, we only consider to tile the upper half triangle where all positions except the diagonal have to be covered twice, as illustrated in Fig. 6. This is done either by overlapping tiles or by counting them twice (using a bit shift). Second, the weight of the tiles is extended to $\{-2,-1,0,1,2\}$. Apart from the multiplier cases 0 (not used) and 1 (used once), we allow to count a tile twice by allowing a weight of 2 (implemented as shift) and we allow negative weights to compensate for overlaps (see M3 in Fig. 7).

Transferring these ideas to the model, the overall multiplier shape $O_{x,y}$ is set to two for all positions above the diagonal, to one on the diagonal and to zero below, i.e.,

$$O_{x,y} = \begin{cases} 1 & \text{when } x = y \\ 2 & \text{when } x > y \\ 0 & \text{otherwise} \end{cases} . \tag{7}$$

To consider the weight of the tiles, we introduce new binary decision variables, that also represent the weight w: $d^{t,w}_{x,y}$ is true when tile t is applied at position (x,y) with weight w. Here, multipliers have to be treated different than squarers: Multipliers have an equal contribution for each position they cover ($o^t_{x,y} = 1$) and their weight is from the set $\{-2,-1,1,2\}$, representing possible bit shifts or subtractions. Squarers have a contribution of one on their diagonal ($o^t_{x,y} = 1$) while their contribution is twice for the remaining bits ($o^t_{x,y} = 2$). Their weights are considered from the set $\{-1,1\}$, to cover their possible subtraction. To model that squarer tiles are only allowed on the diagonal, their weights are forced to 0 when $x \neq y$. Generic multipliers with square shape that are placed on the diagonal are treated like squarers.

Putting all together, constraint C1 is extended as follows:

$$\text{C1'}: \sum_{t=0}^{T-1} \sum_{x'=0}^{x} \sum_{y'=0}^{y} \sum_{w \in W^t_{x,y}} w \, o^t_{x-x',y-y'} \, d^{t,w}_{x',y'} = O_{x,y}$$

$$\text{for } 0 \leq x < w_X, 0 \leq y < w_Y \text{ with } O_{x,y} > 0$$

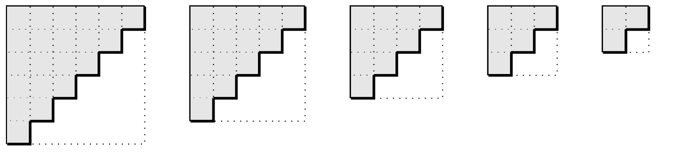


Fig. 8: Utilized logic based squarer tiles.

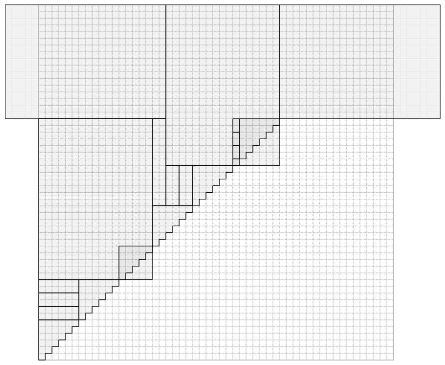


Fig. 9: Optimal tiling of a 53 bit squarer with 4 DSP

with

$$W^t_{x,y} = \begin{cases} \{-1,1\} & \text{when } t \text{ is a squarer and } x = y \\ \{0\} & \text{when } t \text{ is a squarer and } x \neq y \\ \{-2,-1,1,2\} & \text{when } t \text{ is a multiplier} \end{cases} \tag{8}$$

The maximum number of embedded DSP units can be constrained by adding

$$\text{C2}: \quad \#\text{DSP} \geq \sum_{t=0}^{T-1} \sum_{x=0}^{w_X-1} \sum_{y=0}^{w_Y-1} \sum_{w \in W^t_{x,y}} D^t d^{t,w}_{x,y} \; .$$

Within C2, all the individual DSP counts D^t for used tiles (for which $d^{t,w}_{x,y} = 1$) are summed up.

To consider the compression cost of the sign extension when tiles with negative tiling weight $w < 0$ are used, the bits v_i of the sign extension vector are dynamically calculated using

$$\text{C3}: c_{i-1} - 2c_i - v_i + \sum_{t=0}^{T-1} \sum_{x'=0}^{w_X} \sum_{y'=0}^{x'} \sum_{w \in W^t_{x,y}} n^{t,w}_{x',y',i}\, d^{t,w}_{x',y'} = 0$$

for $0 \leq i < w_{\text{out}}$

where $n^{t,w}_{x,y,i}$ defines if a constant sign extension bit is required in column i

$$n^{t,w}_{x,y,i} = \begin{cases} 1 & \text{when } x + y + w^t_{\text{out}} \leq i \wedge w < 0 \\ 0 & \text{otherwise} \end{cases} , \tag{9}$$

$c_i \in \mathbb{Z}$ represents carries in column i and w_{out} and w^t_{out} represent the output word size of the compressor tree and tile t, respectively. The sum of the bits v_i has to be appended to the cost function weighted by 0.65LUT/bit.

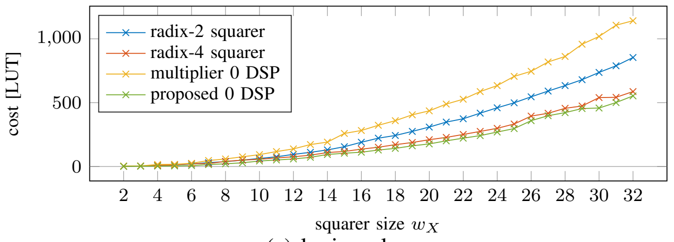
IV. Results

A. Experimental Setup

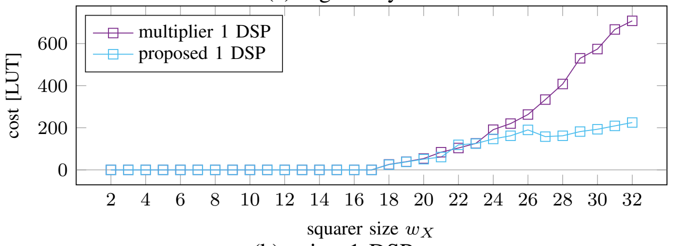
The proposed method was implemented with the help of the VHDL-code generator FloPoCo [2] and is published as open-source [22]. FloPoCo also already provides the reference methods, compressor tree optimization [11] and pipelining framework. The ILP solver Gurobi v9.0.3 was used for the optimizations and synthesis experiments were performed for a Virtex 7 FPGA (xc7k70tfbv484-3) with Xilinx Vivado 2020.1.

B. Test Cases

Synthesis experiments including place and route for squarers with input word sizes in the range $w_X = 2 \ldots 32$ bits were carried out. As reference methods, we used the logic-based radix-2 squarer and the radix-4 squarer proposed in [14] as well as a generic multiplier using the optimal multiplier tiling [20]. For the tiling based methods, the DSP usage was set to either no DSP or one DSP. Timing data was generated by sandwiching the operator between double registers at the inputs and outputs, to ensure more realistic conditions like when it is part of a larger design.

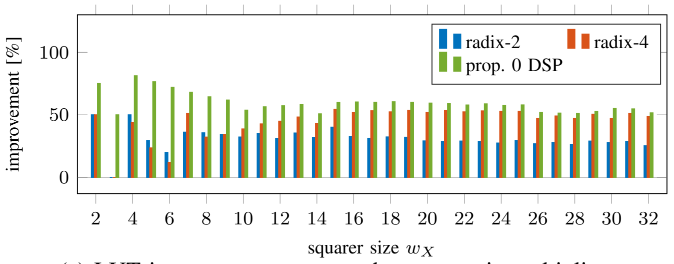


(a) logic only

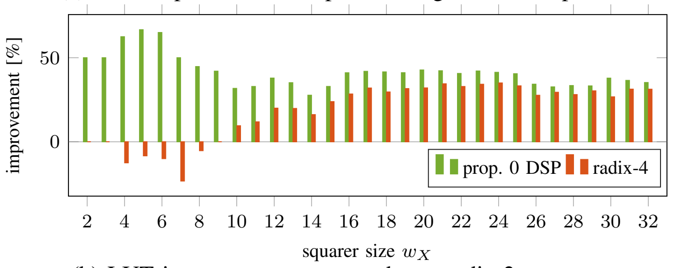


(b) using 1 DSP

Fig. 10: LUT cost of the methods after synthesis



(a) LUT improvement compared to a generic multiplier



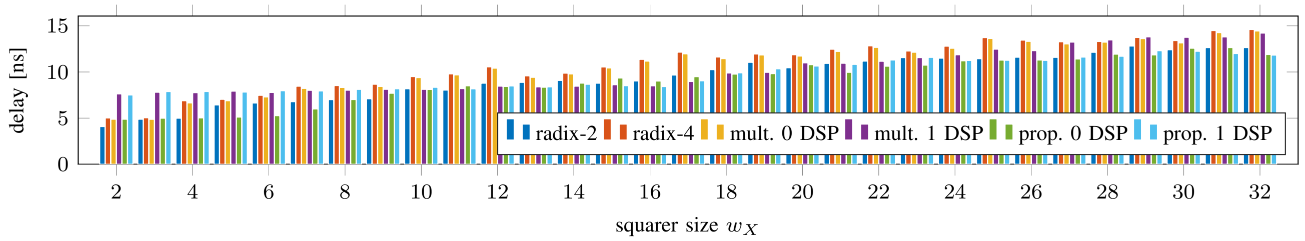
(b) LUT improvement compared to a radix-2 squarer

Fig. 11: LUT improvement for the logic-only squarers

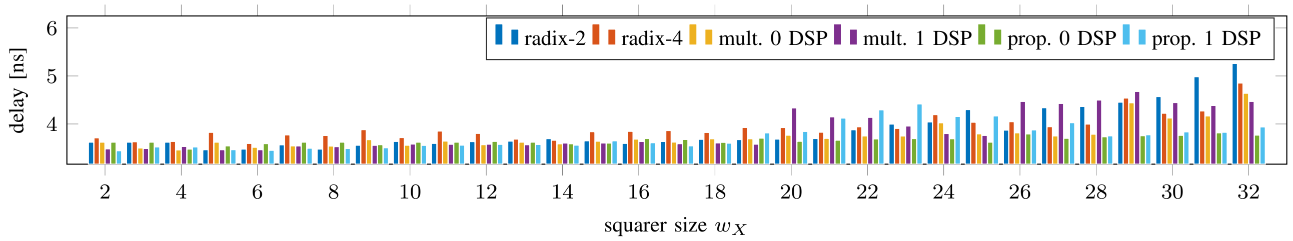
C. LUT Resource Utilization

Fig. 10a shows the synthesis results for the utilized look-up-tables for the logic-only cases. As expected, a logic-only multiplier has the overall highest LUT-costs due to the large redundancies in the partial products, followed by the radix-2 squarer whose AND-gates do not map efficiently to the LUTs on FPGAs. The radix-4 squarer comes close to to the proposed method as the used $2 \times k$ sub-multipliers frequently appear in the proposed tiling solutions. Here, the proposed tiling mainly benefits from the dedicated squarer tiles. In Fig. 10b one DSP is permitted, so the LUT-costs stay zero within the boundaries of the DSP (24×17), i.e., $w_X \leq 17$. Beyond that, significantly less resources are required which diverge down to about one third of the resources.

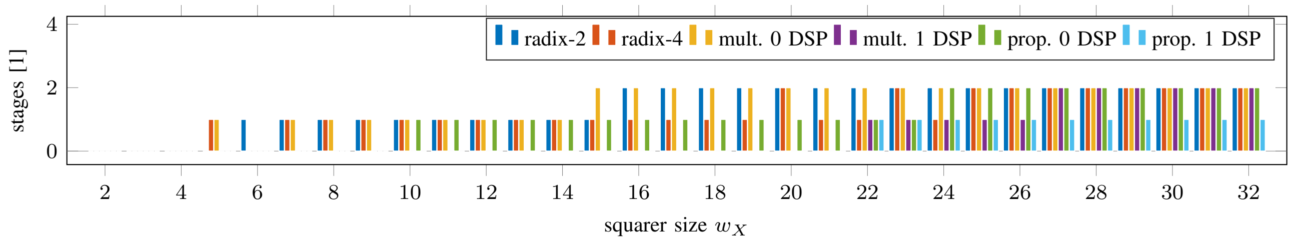
As the differences for the logic-only cases are hard to see in Fig. 10a, several plots showing the relative LUT improvements are given in Fig. 11 Fig. 11a shows that compared to a generic multiplier, typically more than half the LUT resources can be saved. Fig. 11b shows that compared to the radix-2 squarer, the



(a) pure combinatorial



(b) pipelined



(c) Pipeline depths of pipelined circuits

Fig. 12: Critical path delay and pipeline depths for the evaluated methods

radix-4 squarer is worse in the small cases up to $w_X = 10$ bits while the proposed method always provides an improvement over radix-2.

D. Timing Results

Fig. 12 shows the timing results for the evaluated cases. The pure combinatorial case (no pipeline stage at all) is given in Fig. 12a and the pipelined case is shown in Fig. 12b together with the pipeline depth in Fig. 12c. As expected, the delay in the combinatorial case goes up with increasing complexity. Here, the obtained resource reductions also translate to shorter delays, mostly due to less complex compressor trees. It can be observed, that there is a similar upwards trend of the critical path delay in the pipelined case. More complex designs increase the likelihood of an underestimation of the routing delay in contrast to the actual values after place and route during the synthesis process, since the pipelining framework can only rely on average values. The larger squarers could be pipelined more aggressively by providing a larger target frequency to FloPoCo. However, the delay for the proposed method is typically smaller and often required one pipeline stage less.

E. Scalability of the Method

Due to space constraints we limit the extensive evaluation of the results to the cases $w_X = 2 \ldots 32$. The method scales beyond this. A solver timeout was set to 2h as in previous work about multiplier tiling [20], after which the results are still valid but not guaranteed to be optimal. It can be seen in Fig. 13 that the proposed method has a slightly larger runtime

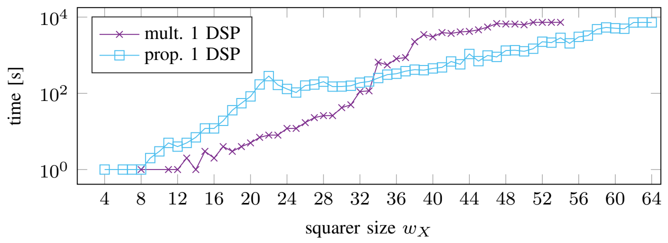


Fig. 13: Solver run-time for the methods

compared to the regular multiplier tiling for small problem sizes due to the almost twofold increase in decision variables considering the different weights.

However, the number of equations is nearly halved, so we can also observe that the proposed method overall scales better to larger problem sizes.

V. Conclusion

This work proposes a flexible method to design resource optimal squarers based on ILP. The model captures design ideas of several previous work, such as the use of DSPs or of efficient radix-4 implementations, and combines them with efficient squarer-specific tiles.

The results show that the proposed tiling based method for squarer design is effective in producing resource optimal solutions for a given number of DSP, compared to the reference methods. Although experiments were conducted on Xilinx Virtex 7 FPGAs, the method can be easily adapted for FPGAs from other vendors like Intel, once target-optimized partial product generator tile designs are provided.

References

[1] D. De Caro, E. Napoli, D. Esposito, G. Castellano, N. Petra, and A. G. Strollo, “Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 5, pp. 1187–1200, 2017.

[2] F. de Dinechin, C. Klein, and B. Pasca, “Generating high-performance custom floating-point pipelines,” in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 59–64.

[3] M. Ercegovac, T. Lang, *Digital Arithmetic*. San Francisco: Morgan Kaufmann Publishers, 2003.

[4] X. Wang, S. Braganza, and M. Leeser, “Advanced components in the variable precision floating-point library.” in *FCCM*. IEEE Computer Society, 2006, pp. 249–258.

[5] F. de Dinechin, M. Joldeş, B. Pasca, and G. Revy, “Multiplicative square root algorithms for FPGAs,” in *Field-Programmable Logic and Applications*, 2010, pp. 574–577.

[6] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Efficient synthesis of compressor trees on FPGAs,” in *Asia and South Pacific Design Automation Conference*, 2008, pp. 138–143.

[7] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Improving synthesis of compressor trees on FPGAs via integer linear programming,” in *2008 Design, Automation and Test in Europe*, 2008, pp. 1256–1261.

[8] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, “Compressor tree synthesis on commercial high-performance FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, p. 39, 12 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2068725

[9] M. Kumm and P. Zipf, “Pipelined compressor tree optimization using integer linear programming,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[10] T. B. Preusser, “Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7.

[11] M. Kumm and J. Kappauf, “Advanced Compressor Tree Synthesis for FPGAs,” *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078 – 1091, 00 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8263391/

[12] Y. Yuan, L. Tu, K. Huang, X. Zhang, T. Zhang, D. Chen, and Z. Wang, “Area optimized synthesis of compressor trees on xilinx fpgas using generalized parallel counters,” *IEEE Access*, vol. 7, pp. 134 815–134 827, 2019.

[13] S. Xu, S. A. Fahmy, and I. V. Mcloughlin, “Efficient large integer squarers on fpga,” in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 198–201.

[14] B. Lee and N. Burgess, “Improved small multiplier based multiplication, squaring and division,” in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, 2003, pp. 91–97.

[15] S. Gao, N. Chabini, D. Al-Khalili, and J. M. P. Langlois, “FPGA-Based Efficient Design Approaches for Large Size Two’s Complement Squarers,” *Journal of Signal Processing Systems*, vol. 58, no. 1, pp. 3–15, 2008.

[16] F. de Dinechin and B. Pasca, “Large multipliers with fewer DSP blocks,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 250–255.

[17] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, “Multipliers for floating-point double precision and beyond on FPGAs,” *SIGARCH Comput. Archit. News*, vol. 38, no. 4, p. 73–79, Jan. 2011. [Online]. Available: https://doi.org/10.1145/1926367.1926380

[18] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, “Arithmetic core generation using bit heaps,” in *International Conference on Field programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[19] A. Boettcher, K. Kullmann, and M. Kumm, “Heuristics for the design of large multipliers for FPGAs,” in *Symposium on Computer Arithmetic (ARITH)*, 2020, pp. 17–24.

[20] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, “Resource optimal design of large multipliers for FPGAs,” in *Symposium on Computer Arithmetic (ARITH)*, 2017, pp. 131–138.

[21] A. Boettcher, M. Kumm, and F. de Dinechin, “Resource optimal truncated multipliers for FPGAs,” in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2021.

[22] “Project page FloPoCo.” [Online]. Available: http://www.flopoco.org/