A Combined Arithmetic-High-Level Synthesis Solution to Deploy Partial Carry-Save Radix-8 Booth Multipliers in Datapaths

Alberto A. Del Barrio, *Member, IEEE*, Román Hermida, *Senior Member, IEEE*, and Seda Ogrenci-Memik, *Senior Member, IEEE*

***Abstract*— While partial carry-save adders are easily designed by splitting them into several fragments working in parallel, the design of partial carry-save multipliers is more challenging. Prior approaches have proposed several solutions based on the radix-4 Booth recoding. This technique makes it possible to diminish the height of a multiplier by half, this being the most widespread option when designing multipliers, as only easy multiples are required. Larger radices provide further reductions at the expense of the appearance of hard multiples. Such is the case of radix-8 Booth multipliers, whose critical path is located at the generation of the 3X multiple. In order to mitigate this delay, in our prior works, we proposed to first decouple the 3X computation and introduce it in the dataflow graph, leveraging the available slack. Considering this, we then present a partial carry-save radix-8 Booth multiplier that receives three inputs in this format, namely, the multiplicand, the multiplier, and the 3X multiple. Moreover, the rest of the datapath is adapted to work in partial carry-save. In comparison with conventional radix-4 and radix-8 Booth-based datapaths, the proposal is able to diminish the execution time and energy consumption while benefits from the area reduction provided by the selection of radix 8. Furthermore, it outperforms prior state-of-the-art partial carry-save multipliers based on radix 4.**

***Index Terms*— Multipliers, Booth, radix 8, partial carry-save, slack.**

I. Introduction

MULTIMEDIA, signal processing applications and others are usually dominated by integer addition and multiplication [1]–[4]. Hence, developing faster, smaller and power efficient Functional Units (FUs) is critical to implement efficient circuits. The fastest adders, such as the Kogge-Stone prefix adder [5], [6], possess a noticeable area and power penalty. Several approximate adders have appeared that are aimed at improving these paramenters, especially by reducing the average delay [7]–[10] and offering different area/power tradeoffs. Additive structures have also been optimized by using redundant formats such as the total [11] or partial carry-save adders [12].

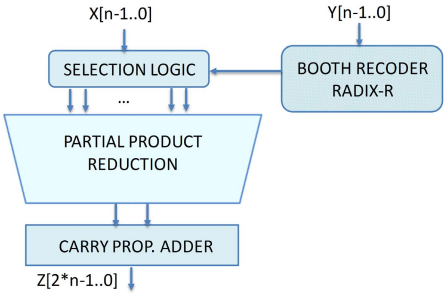

Fig. 1. Radix R Booth multiplier.

Nevertheless, datapaths are still dominated by multipliers, as their delay, area and power grow faster than in the case of adders [13]–[16]. Multipliers typically consist of an additive tree known as the partial product matrix (PPM), which accumulates the partial products and reduces them to just two operands, and a last stage Carry Propagate Adder (CPA), which adds these two operands and calculates the final result. Given an $m \times n$ multiplier, the PPM is composed of $m \times n$ 1-bit partial products $p_{i,j}$, defined by Equation 1.

$$p_{i,j} = x_i y_j, \quad \forall i, j, \ 0 \leq i < m, \ 0 \leq j < n. \tag{1}$$

where, $X = x_{m-1}x_{m-2}\ldots x_1x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_1y_0$ are the multiplicand and the multiplier, respectively. Hence, in order to reduce the complexity of a multiplier, either the width m or the height n must be diminished. Lowering m leads to truncated multipliers [17]–[19], which is not the purpose of this work. Adopting the other approach, the height of the PPM is usually reduced by applying a Booth recoding [14], [15], [20] in radix $R = 2^r, r > 0$, which maintains the accuracy of the multiplier. The generic structure of a Radix R Booth Multiplier is depicted in Figure 1 [14], [15], [20]. Thus, the height of an $m \times n$ multiplier is reduced from n to $\lceil (n+1)/r \rceil$. Intuitively, the larger r is, the better. Nevertheless, applying this recoding technique incurs a certain area penalty to calculate the *Booth multiples* or subproducts. For this reason, typically $r = 2$ [21]–[23], because given a product X*Y, only $\pm$ 0X, $\pm$ 1X and $\pm$ 2X multiples need to be generated [20]. As can be observed, all of them can easily be

Manuscript received February 19, 2018; revised June 4, 2018 and July 20, 2018; accepted August 16, 2018. This work was supported in part by EU (FEDER) and the Spanish MINECO under Grant TIN 2015-65277-R, in part by the UCM-Banco Santander under Grant PR26-16/20B-1, and in part by the NSF CCF-1422489 Grant. This paper was recommended by Associate Editor A. Cilardo. *(Corresponding author: Alberto A. Del Barrio.)*

A. A. Del Barrio and R. Hermida are with the Department of Computer Architecture and Automation, Computer Science Faculty, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: abarriog@ucm.es; rhermida@ucm.es).

S. Ogrenci-Memik is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: seda@eecs.northwestern.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2018.2866172

TABLE I
RADIX 8 BOOTH RECODER FOR A BOOTH TUPLE $t_p = y_{3*p+2}y_{3*p+1}y_{3*p}y_{3*p-1}$

y_{3*p+2}	y_{3*p+1}	y_{3*p}	y_{3*p-1}	Select	Select Enc	Sign	y_{3*p+2}	y_{3*p+1}	y_{3*p}	y_{3*p-1}	Select	Select Enc	Sign
0	0	0	0	+0X	000	0	1	0	0	0	-4X	100	1
0	0	0	1	+1X	001	0	1	0	0	1	-3X	011	1
0	0	1	0	+1X	001	0	1	0	1	0	-3X	011	1
0	0	1	1	+2X	010	0	1	0	1	1	-2X	010	1
0	1	0	0	+2X	010	0	1	1	0	0	-2X	010	1
0	1	0	1	+3X	011	0	1	1	0	1	-1X	001	1
0	1	1	0	+3X	011	0	1	1	1	0	-1X	001	1
0	1	1	1	+4X	100	0	1	1	1	1	-0X	000	1

calculated via shift and negation operations. For $r > 2$, there appear *hard multiples*, i.e., those that are not composable with just shifts and negations. Due to their calculation, the penalty then exceeds the gains from the reduction in the PPM height. In particular, the case of the radix 8 Booth recoding requires the computation of the following multiples: $\pm 0X$, $\pm 1X$, $\pm 2X$, $\pm 3X$ and $\pm 4X$. The $\pm 3X$ multiples are not straightforward to compute. Typically, $3X$ is computed as the addition of $2X+X$, thus increasing the critical path of the multiplier. Previous approaches have proposed decoupling this computation by just pipelining [24] or by inserting the $3X$ computation as an independent operation in the Dataflow Graph (DFG) by leveraging the slack [25]. In this way it is possible to take advantage of the larger height reduction enabled by the radix 8 recoding instead of the radix 4 recoding, without increasing the radix 8 Booth multiplier's critical path. This paper is based on the latter approach. In this work the partial carry-save format is incorporated into the whole datapath, which will imply modifying both the $3X$ calculation unit and the multiplier. A detailed study of the relationship between the radix and the fragment size is presented and applied to the case of radix 8. Therefore, it is possible to develop a methodology to deploy datapaths that work entirely in partial carry-save format and just reduce results to a non-redundant form when they are output nodes of the DFG. According to the synthesis results, our proposed multipliers outperform conventional radix 4 and radix 8 designs, as well as our prior radix 4 partial carry-save multipliers [26]. Furthermore, the datapaths implemented with the proposed approach can reduce 27% energy consumption, being 22% faster on average as well.

The rest of the paper is organized as follows: Section II examines the state of the art of Booth-based multipliers. Section III justifies the use of radix 8 instead of radix 4 and shows the potential benefits of decoupling the $3X$ multiples. Besides, at the end of this section, the problem presented in this paper is clearly described. Section IV presents two concepts that are essential in understanding this paper: how the $3X$ nodes insertion algorithm works; and the radix 4 Booth partial carry-save multiplier, which is taken as reference. Section V describes the implementation of the partial carry-save units to deploy datapaths using this format, and their customization for the case of radix 8. Section VI illustrates the use of the proposed radix 8 Booth partial carry-save multiplier. Sections VII and VIII discuss our experimental results and contain our concluding remarks. Finally, an appendix has been included to provide the theoretical foundations that justify the design decisions taken in this paper.

II. RELATED WORK

Table I shows the truth table for the radix 8 Booth encoding [14], [15], [20]. As can be observed, every four bits corresponding to $t_p = y_{3*p+2}y_{3*p+1}y_{3*p}y_{3*p-1}$ produce two outputs, namely: the selection bits, labelled as $SelectEnc$, and the sign bit, labelled as $Sign$. In the following, the t_p bit groups shall be referred to as *Booth tuples*. As can be observed, there are tuples producing $\pm 3X$ hard multiples, which causes an additional delay when computing $3X$, typically as $2X + X$.

The proposals presented in [27] and [28] describe a hybrid architecture combining both radix 4 and 8. The first partial products are generated using radix 4 recoding, while the $3X$ computation is taking place, and the latter ones are generated using radix 8 recoding. On the other hand, in the works presented in [29]–[31], authors try to optimize constant multipliers. Another type of proposal is described in [32], where a Redundant Signed-Digit Booth encoding technique is introduced to remove the hard multiples, at the expense of duplicating the PPM. With the Redundant Signed-Digit system, every bit x_i is represented by two bits x_i^+ and x_i^-, such that $x_i = x_i^+ - x_i^-$. The main advantage is that addition becomes carry free, but in the multiplier a PPM will be necessary to accumulate the multiples from the x_i^+ bits, and another one for the multiples generated by x_i^-. This problem also appears in the integer multiplier embedded within the Floating-Point unit proposed in [33] and [34]. Some authors have also proposed the use of speculative techniques to accelerate the multiplication, but without considering Booth recoding [35].

In order to reduce the aforementioned $3X$ delay in radix 8 Booth multipliers, Jiang *et al.* [36] propose an adder built from 2-bit approximate adders, but this suffers from scalability and inaccuracy issues. Other works have proposed a partial carry-save addition [37], [38] for computing this hard multiple [39]. In partial carry-save mode an operand is still computed as $X = U + A$, U and A being the result and carries vector, respectively. But unlike total carry-save only certain positions in A may contain a '1', i.e. every k-bits [26], [37], [39]. This makes it possible to reduce the $3X$ computation time. Some previous works have also dealt with partial redundant multipliers, but always considering the classical radix 4 Booth encoding. The work in [37] describes a multiplier that is able to work with partial carry-save multiplicands and multipliers. Given n and k word and fragment bitwidths, respectively, every multiple generated after Booth encoding requires $n+n/k$ bits, plus an extra '1' for negative multiples, which is translated into a larger 4:2 compression tree. Daumas and Matula [40] and Tsoumanis *et al.* [41] propose a precoder stage to deal

with both redundant and non-redundant multipliers, but not multiplicands. The work in [42] describes a multiplier that is able to receive both inputs in carry-save representation, with a complex multiples generation stage, which produces a noticeable area increase. The work in [38] presents a design flow utilizing the previous multiplier. Finally, in our previous work [26] we presented a partial carry-save multiplier that is able to deal with both inputs in this format with a negligible overhead, and does this by correcting both the multiplicand and the multiplier on-the-fly. Inside this multiplier, the following actions take place prior to generating the multiples:

- The multiplicand is calculated using the Carry-Select technique [14], [15], i.e. each k-bit fragment is calculated with both '0' and '1' carry-in values while the real carries are calculated (in just $log(n/k) - 2$ levels).
- After the real carries of both inputs are computed in $log(n/k) - 2$ levels, the multiplier is recoded using two encodings, namely: Booth_0, or conventional Booth, and Booth_1, which assumes that a '1' must be added to the $2 * p^{th}$ position of each tuple. It should be noted that radix 4 is employed, so the tuple is actually a triplet $t_p = y_{2*p+1}y_{2*p}y_{2*p-1}$.

Once the real carries are properly computed, the correct multiplicand fragments are selected, as well as the correct encodings for the multiplier. In this paper we now propose to apply a similar technique to multiply two numbers in partial carry-save format, but using radix 8. In order to overcome the delay overhead imposed by the $3X$ multiple calculation, it will first be decoupled from the multiplier and inserted in the DFG as an independent operation by leveraging the datapath slack [25]. Furthermore, the unit responsible for computing the $3X$ operation will also be adapted to the partial carry-save format. Therefore, the proposed multiplier will receive three inputs in partial carry-save format, namely: the multiplicand (X), the multiplier (Y) and the $3X$ multiple. A detailed study of the theoretical foundations and the relationship between the radix and the fragment size are presented in the appendix.

III. Radix 8 vs Radix 4 Booth Multipliers

In order to justify this work, several multipliers were synthesized using *Synopsys Design Compiler* with a 65 nm target-library and without placing any delay constraint. We measured the delay, area, power and energy of the radix 4 and radix 8 Booth multipliers. Both types of multipliers implement the PPM stage through a 4:2 compressor tree and the CPA stage through a Kogge-Stone adder. Table II presents the delay, area, power, and energy of the radix 8 Booth multipliers after synthesis. All the values have been normalized with respect to a radix 4 Booth multiplier for different data widths. It should also be noted that two values are presented for each metric that is reported, namely: those considering that the $3X$ multiple is being calculated within the radix 8 Booth multiplier, labelled as *w/3X*, and those considering that the $3X$ multiple has been precalculated in the datapath as an independent operation, labelled as *w/o 3X*.

As can be observed, in terms of delay, radix 8 implementations are slightly faster when the size is small, i.e. 8 and 16-bits. However, for 32-bits and larger sizes, radix 4 implementations are better. As shown in column *w/o 3X*, the delay can be balanced and even shortened by precalculating the $3X$ multiple for radix 8 implementations. Regarding the area and power, it is clear that reducing the PPM from $\lceil (n+1)/2 \rceil$ to $\lceil (n+1)/3 \rceil$ achieves an improvement. As shown in Table II, the reduction in area ranges from 34% to 24% for complete radix 8 Booth multipliers, and from 40% to 25% when precalculating the $3X$ multiple. In the case of power, the reduction ranges from 16% to 9% for complete radix 8 Booth multipliers, and from 28% to 18% when precalculating the $3X$ multiple. In terms of energy, radix 8 implementations are only more efficient for small bitwidths. Nevertheless, this issue can also be resolved, actually obtaining energy reductions, by precalculating the $3X$ multiple.

TABLE II
Radix 8 Booth Multipliers Synthesis Results Normalized With Respect to Radix 4 Booth Multipliers

	Delay		Area		Power		Energy	
n	w/3X	w/o 3X	w/3X	w/o 3X	w/3X	w/o 3X	w/3X	w/o 3X
8	0.85	0.85	0.66	0.60	0.84	0.72	0.71	0.61
16	0.91	0.87	0.73	0.69	0.86	0.75	0.79	0.65
32	1.13	0.91	0.72	0.70	0.89	0.79	1.00	0.72
64	1.13	1.01	0.75	0.74	0.89	0.81	1.01	0.81
128	1.26	0.96	0.76	0.75	0.91	0.82	1.14	0.79

All the aforementioned values have been computed when considering just an individual instance of multiplier, so including the decoupled $3X$ computations and routing them will contribute to increasing the area and energy consumption of the datapath. Nonetheless, this example illustrates two main ideas: firstly, there is a delay reduction in the radix 8 multiplier when precalculating the $3X$ multiple; and, secondly, moving to radix 8-based implementations produces a noticeable area and energy reduction with respect to the radix 4-based implementations. Therefore, these gains can be leveraged to route and store the $3X$ computations.

A. Problem Statement

Partial carry-save formats have proved to be efficient if we look at the state of the art. Hence, partial carry-save multipliers are essential to implement datapaths using this format. All the solutions proposed in literature [26], [37], [38], [40]–[42] are based on radix 4 Booth recoding. Nevertheless, a larger radix may provide greater benefits in terms of energy if we are able to efficiently deal with the hard multiples. Therefore, we propose a method for doing this, as well as a generic Booth-based multiplier for partial carry-save inputs and any 2^r radix. In this paper, we apply this method to the specific case of radix 8 and explore different fragment sizes.

IV. Preliminaries

In this section, an overview of the basic techniques that constitute the foundations of this work will be given. First, an example of strategic insertion of 3X calculations into the datapath will be presented. Second, the basic principles of the On-the-fly Correcting Radix 4 Multispeculative Multiplier (OMSM-B4) will be illustrated.

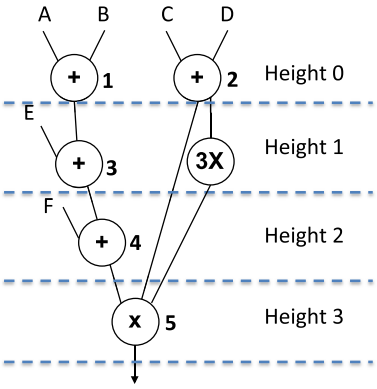

Fig. 2. Example of $3X$ node insertion [25].

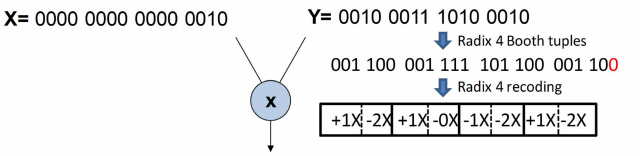

Fig. 3. A multiplier with non-redundant operands and radix 4 Booth encoding.

A. Inserting the 3X Nodes

In order to decouple the $3X$ calculation, we introduce an extra $2X + X$ addition node per product, selecting the farthest predecessor, i.e. X, of the multiplication in terms of DFG height. In this way, the probability of providing enough slack to schedule the $2X + X$ operations is maximized. In other words, the probability of increasing the latency is minimized. Figure 2 illustrates how this approach works. First, it should be noted that each operation is labelled by a number located at the right of the node. In this figure it can observed that if the $2X+X$ node were inserted after **Operation 4**, this would incur additional latency. However, as shown in Figure 2, selecting **Operation 2** is the best choice, as its distance to **Operation 5** is greater than the distance between **Operations 4** and **5**. Or, in other words, the height of **Operation 2** is less than the height of **Operation 4**. Thus, by selecting the operand supplied by the output of **Operation 2** to pre-calculate $3X$ $(2X + X)$, the latency of the final schedule is not affected.

B. The On-the-Fly Correcting Radix 4 Booth Multispeculative Multiplier

In this subsection, the foundations of the radix 4 partial carry-save Booth multiplier presented in [26] will be outlined through the example shown in Figures 3 and 4. Figure 3 displays an example of multiplication utilizing radix 4 Booth encoding, while Figure 4a depicts a case with the same input operands but using partial carry-save format. In this example, 16-bit FUs with fragments of $k = 4$-bits have been considered. This means that there will be $16/4 = 4$ fragments producing *group generate* (Gx, Gy) and *propagate* (Px, Py) signals. It should be mentioned that the group generate signals are actually the carry-out signals produced by the fragments. The generate and propagate signals produced by the most significant fragment have been labelled as '-', because they will no longer be utilized.

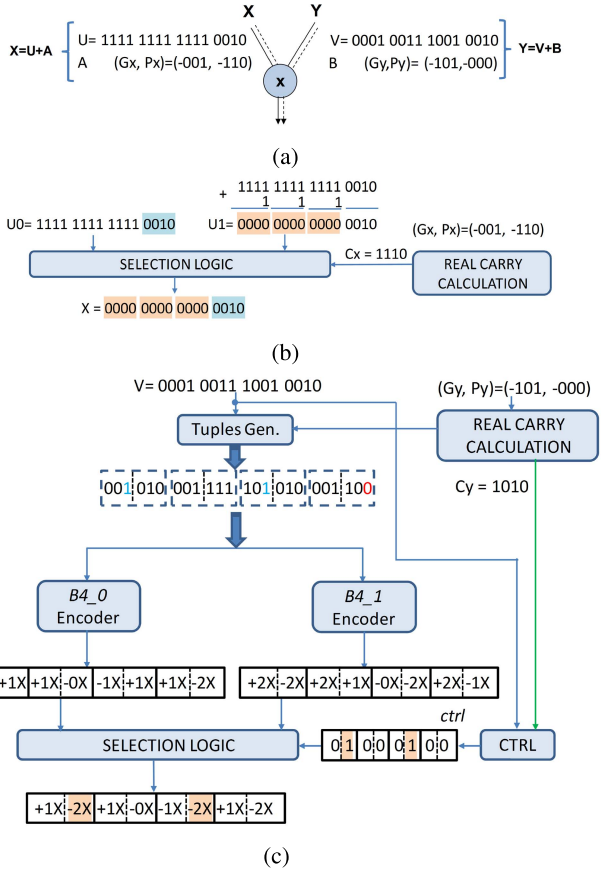

Fig. 4. On-the-Fly Correcting Radix 4 Booth Multispeculative Multiplier example of use. (a) A multiplier with partial carry-save operands, $k = 4$. (b) Reconstructing the non redundant multiplicand. (c) Multiplier branch: radix 4 carry select Booth encoding generates the multiples.

Given a multiplicand in partial carry-save format, i.e. $X = U + A$, Figure 4b displays how its non-redundant version is calculated. For this purpose, the carry-select technique is employed. In other words, all the fragments are computed in parallel with both carry-in '0' (there is no actual addition) and '1', while the actual carries of the fragments are computed by the Real Carry Calculation unit [26], [43]. This unit is a crucial part of this step, as the carries generated by each fragment of a partial carry-save adder might be incorrect. However, the Gx, Px group signals generated by every fragment allow the fast computation of the true carries. The selection logic simply chooses a fragment belonging to either $U0$ or $U1$, according to the values of Cx. It should be noted that $U0 = U$, while $U1$ is obtained after adding '1' to every fragment belonging to U, except the first one.

Figure 4c provides an overview of the multiplier branch. Given a multiplier in partial carry-save format, i.e. $Y = V + B$, the V signal will be used to generate the Booth encoding. First, a Real Carry Calculation unit computes the actual carries of the multiplier using the Gy, Py signals. Second, tuples generation takes place, which is not as immediate as in a conventional multiplier. As the non-redundant multiplier is not known, the value of some multiplier bits needs to be calculated. This calculation requires few logic levels, as it just depends on the true carries and some bits belonging to V. After the tuples are generated, the B4_0 encoder implements the

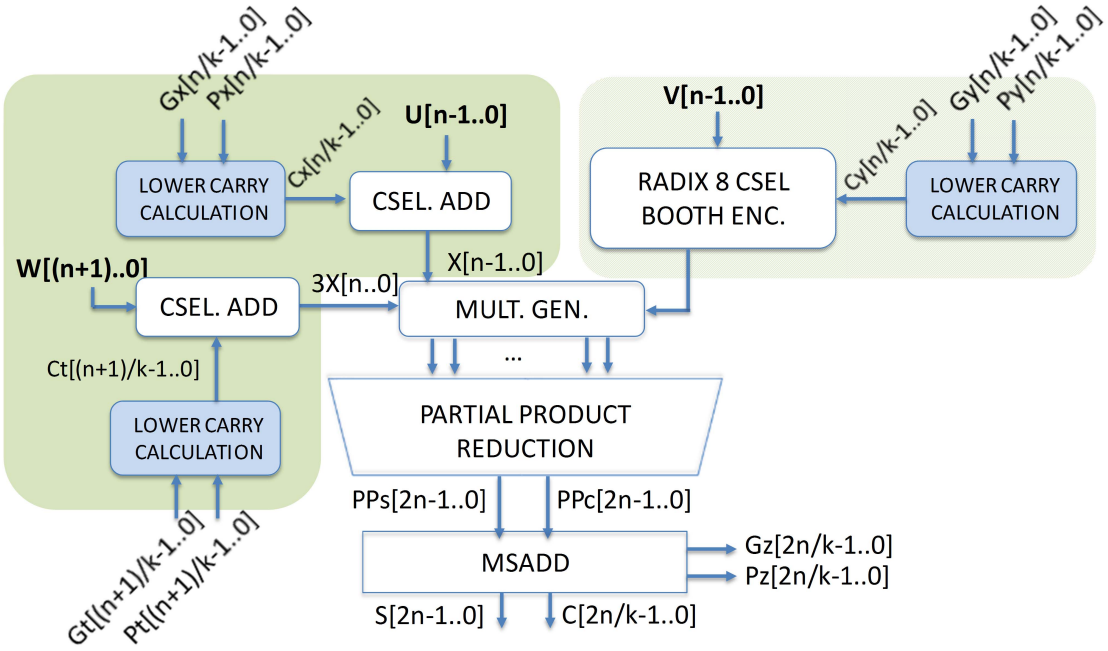

Fig. 5. On-the-fly Correcting Radix 8 Booth Multispeculative Multiplier Architecture (OMSM-B8).

typical radix 4 recoding, while the B4_1 encoder implements the same recoding but considering that every Booth tuple $t_p = v_{2*p+1}v_{2*p}v_{2*p-1}$ has a carry-in signal equal to '1' in the $2*p$ position. In other words, this is equivalent to adding the constant "010" to every tuple. Nevertheless, instead of adding and then recoding, the encoding is directly performed following a truth table [26], which is faster. Finally, the correct encoding for every tuple will be selected according to the control ($ctrl$) signals. As can be observed, the final encoding coincides with that provided by the conventional radix 4 Booth recoding shown in Figure 3. The main advantage of this approach is that all the fragments work in parallel once the real carries are known.

After generating the correct Booth encoding as well as the multiplicand (i.e. non-redundant format), Booth multiples generation and PPM addition take place as in conventional multipliers. Finally, the CPA stage is performed via a Multi-speculative Adder [10], generating a result in partial carry-save format, which may be used by other partial carry-save units.

V. Partial Carry-Save Functional Units

Besides the algorithmic support for including the $3X$ computations in the DFG, it is necessary to utilize specific FUs to enable the datapath to work in partial carry-save format. In order to deal with radix 8 subproducts, the multiplier design presented in [26] will be modified. Moreover, a new FU named *Multispeculative Tripler* (MST) will be implemented in order to compute $3X$ in a partial carry-save fashion as well.

A. The On-The-Fly Correcting Radix 8 Booth Multispeculative Multiplier

Figure 5 depicts the architecture of the proposed radix 8 Booth Multispeculative Multiplier (OMSM-B8). Provided that $X = U + A$, $Y = V + B$ and $T = 3X = W + D$, where U, V and W are result vectors and A, B and D are carry vectors, this unit is responsible for computing $Z = X * Y = S + C$ by applying a radix 8 Booth recoding techique, where S is a result vector and C is a carries vector. In other words, the OMSM-B8 receives two inputs and the $3X$ in partial carry-save format, and produces an output in partial carry-save form as well.

As can be observed, first the real carry values are computed by using the generate and propagate k-bit group signals in the *Lower Carry Calculation* blocks. This is performed in the same fashion as in [26], where the carries calculation is split by performing the first two levels in the adders that partially compute X, Y and T. Splitting the Carry Calculation blocks into Upper (in the prior adders) and Lower (in the own multiplier) modules, implies that in some cases the Lower modules do not even exist, as $max(0, \lceil log(n/k) \rceil - 2)$ levels are required to generate the real carries (e.g. $n = 16$ and $k = 6$). In the multiplicand branch, indicated by a dark green background, the actual values of X and $3X$ are correctly computed in parallel with a Carry-Select Adder (*CSEL. ADD*). Once the corresponding real carries are known, the proper fragments are selected to drive X and $3X$ to the multiples generation unit (*MULT. GEN*). In the multiplier branch, indicated by a light green background, the radix 8 Booth encoding takes place, utilizing a Carry-Select-based technique as in [26]. Two encodings are produced, namely: the radix 8 $Booth\_0$ ($B8\_0$) and $Booth\_1$ ($B8\_1$). The $B8\_0$ is equivalent to a conventional radix 8 Booth recoding, while the $B8\_1$ encodes the p^{th} tuple $t_p = v_{3*p+2}v_{3*p+1}v_{3*p}v_{3*p-1}$ by considering the alternative tuple $t_p +$"0010". Once the real carries are known, the proper recoding is selected and driven to the multiples generation unit. All the multiples are generated and reduced by using a 4:2 compressor tree, and finally summed with a Multispeculative Adder (MSADD) [10], which produces the result in partial carry-save format.

TABLE III

$B8\_1$ RADIX 8 BOOTH RECODER FOR A BOOTH TUPLE $t_p = y_{3*p+2}y_{3*p+1}y_{3*p}y_{3*p-1}$

y_{3*p+2}	y_{3*p+1}	y_{3*p}	y_{3*p-1}	Select	sel_p	$sign_p/msb_p$	c_p	y_{3*p+2}	y_{3*p+1}	y_{3*p}	y_{3*p-1}	Select	sel_p	$sign_p/msb_p$	c_p
0	0	0	0	+1X	001	0	0	1	0	0	0	-3X	011	1	0
0	0	0	1	+2X	010	0	0	1	0	0	1	-2X	010	1	0
0	0	1	0	+2X	010	0	0	1	0	1	0	-2X	010	1	0
0	0	1	1	+3X	011	0	0	1	0	1	1	-1X	001	1	0
0	1	0	0	+3X	011	0	0	1	1	0	0	-1X	001	1	0
0	1	0	1	+4X	100	0	0	1	1	0	1	-0X	000	1	0
0	1	1	0	-4X	100	1	0	1	1	1	0	+0X	000	0	1
0	1	1	1	-3X	011	1	0	1	1	1	1	+1X	001	0	1

Finally, it should be noted that the generate and propagate vectors, as well as the real carries, possess one additional position in comparison with [26]. The reason is that in our previous work n was always a multiple of k, so the last generate and propagate signals could be discarded as the most significant carry-out was not necessary to select a fragment result within the CSEL adders. However, in this work n will not necessarily be a multiple of k, and an additional bit will be required.

1) The Radix 8 Carry Select Booth Encoder: A radix 8 Booth recoder processes the multiplier operand Y in groups of 4-bits in parallel, overlapping the most significant bit of every group with the least significant of the following one, and produces the selection and the sign bits. The sign bit is used to compose the negative partial products. This bit is '0' when the partial product has a positive weight and '1' otherwise.

The original recoder works directly with the multiplier Y. However, in order to deal with partial carry-save multipliers we apply the encoding to V, with $Y = V + B$. Using k-bit fragments, the real carry-in signals to every fragment are unknown until they have been computed by the Y Real Carry Calculation unit. Two recoders working in parallel are required then: one considering a '0' carry-in ($B8\_0$) and another considering a '1' carry-in ($B8\_1$). Hence, our recoder shall be named Radix 8 Carry Select Booth Encoder (B8CSBE).

Figure 6 depicts the structure of the generic cell responsible for generating the B8CSBE recoding for a given tuple $t_p = v_{3*p+2}v_{3*p+1}v_{3*p}v_{3*p-1}$. In addition to both recoders, some extra units are necessary to properly complete the encoding process. As indicated in [26], first is necessary to generate the actual Booth tuple (t'_p), as this may present a different least significant bit (*lsb*) because the $B8\_1$ recoding of the prior cell may generate a different most significant bit (*msb*). The calculation of the actual most significant bit (msb_p) is handled by the Most Significant Bit Unit (*MSBU*). Moreover, there can also be internal carries (c_p) within every fragment due to the $B8\_1$ recoding. Thus, besides the multiple select bits (sel_p) and the multiple sign bit ($sign_p$), Table III defines the actual most significant bit (msb_p) as well as the internally generated carry (c_p). As can be seen, $sign_p$ and msb_p are in fact the same signal, because both are determined by the actual msb of the Booth tuple.

Hence, besides the tuple $t_p = v_{3*p+2}v_{3*p+1}v_{3*p}v_{3*p-1}$, every cell belonging to fragment i receives the following signals as inputs:

- msb_{p-1}. This is the msb proceeding from the previous cell $(p-1)$. As has been mentioned, it is necessary for selecting the actual lsb of the current cell (p).

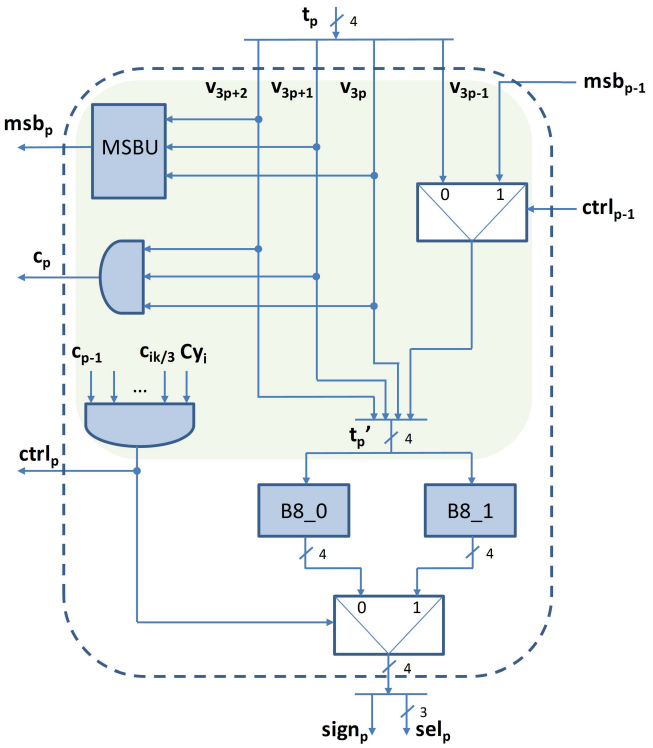

Fig. 6. Radix 8 Carry Select Booth Encoder (B8CSBE) cell.

- Cy_i. This is the real carry of the i^{th} fragment, computed by the Y Real Carry Calculation unit.
- Each internal carry belonging to fragment i. According to the appendix, these signals are: $c_{ik/3}$, $c_{ik/3+1}, \ldots, c_{p-1}$. It should be noted that k will be a multiple of 3 as radix 8 is being considered.
- $ctrl_{p-1}$. This is the control signal generated by the previous cell $(p-1)$. It is required to select the actual lsb of the current cell (p). Thus, if the $(p-1)^{th}$ cell selects the $B8\_1$ encoding, i.e. $ctrl_{p-1}$='1', the actual lsb of the p_{th} cell will be msb_{p-1}.

Analogously, besides the recoding ($sign_p$ and sel_p), every cell belonging to fragment i generates the following signals as outputs:

- msb_p. This is the actual msb generated by the current cell (p). It is critical to observe that this signal only depends on v_{3*p+2}, v_{3*p+1} and v_{3*p}, so possible changes of the lsb never ripple to the following cell.
- c_p. This is the internal carry generated by the $B8\_1$ recoding in the current cell (p). The same observation as in the case of msb_p must be made: it is the logic

AND of v_{3*p+2}, v_{3*p+1} and v_{3*p}, and is independent of the lsb.

- $ctrl_p$. This is the control signal generated by the p^{th} cell. It will be utilized to decide which encoding is employed in the current cell (p), as well as to determine the lsb of the following cell ($p+1$). This signal is implemented with a logic AND possessing a variable number of inputs. In practice, k will be small and there will never be more than 3 tuples within a fragment. In other words, the maximum number of inputs of this AND will be 3.

Therefore, this cell accomplishes two objectives: on the one hand generating the actual tuple t'_p using the logic enclosed within the darkened area in Figure 6; and, on the other hand, producing the proper recoding of the multiplier operand. The formal proof of the equations that generate the aforementioned signals is given in the appendix.

2) Tuples Alignment: In addition to the problems imposed by the partial carry-save format of the operands, the radix 8 recoding presents an extra challenge: the alignment of the tuples. Definition 1 formalizes this concept.

Definition 1: Two radix $R = 2^r$ Booth tuples t_p and t_q are aligned iff any carry to be added to t_p and t_q is added in positions i, j such that $i \ mod \ r = j \ mod \ r$, i.e. $i_r \equiv j_r$.

In our previous work, the recoder was developed in combination with fragments whose size k was a power of 2 [26], which in radix 8 leads to complicating the design. This fact is illustrated in Figure 7a, where k = 4-bit is considered. Red arrows drive the real carries to the fragments, while the blue dashed arrows drive the internal carry-in signals. The most significant bits of the tuples are surrounded by a solid green box. For instance, let us consider tuple $t_2 = v_8v_7v_6v_5$. Bit v_6 is affected by the internal carry c_1, while bit v_8 is affected by the real carry Cy_2. Hence, depending on the values of these two signals, four cases are possible for properly recoding this tuple. In the case of tuple $t_3 = v_{11}v_{10}v_9v_8$ this happens in different positions, namely v_8 and v_9. Nevertheless, this does not happen in Figures 7b and 7c, which consider k = 3-bit and k = 6-bit fragments. In these cases, given a tuple $t_p = v_{3*p+2}v_{3*p+1}v_{3*p}v_{3*p-1}$ the real/internal carry values are always added to the q^{th} position, where $q = 3*s$. In other words, all the tuples are *aligned*, unlike in Figure 7a. The appendix of this paper presents the theoretical basis for this design decision, and provides a proof that the tuples are aligned iff $k \ mod \ r = 0$. Therefore, as radix 8 is being employed, in order to align the tuples the rest of the paper shall consider fragment sizes which are a multiple of 3.

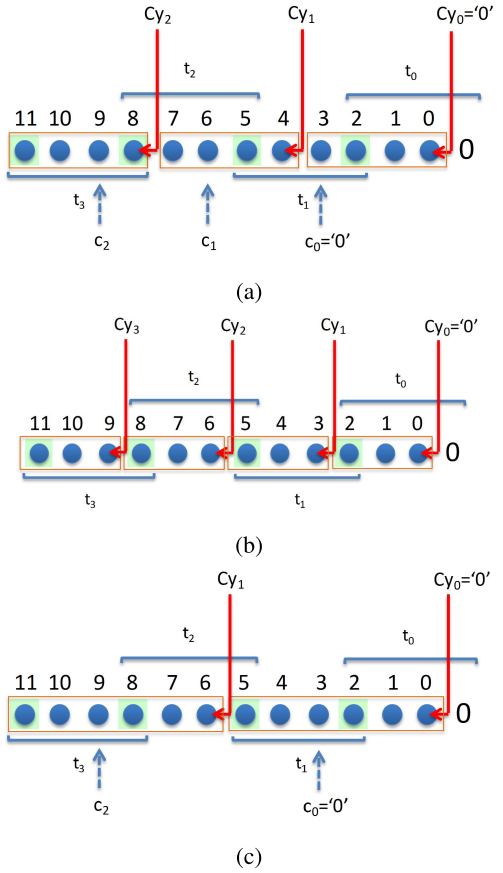

Fig. 7. Radix 8 Carry Select Booth Encoder alignment issue. (a) B8CSEL with k = 4-bit fragments. (b) B8CSEL with k = 3-bit fragments. (c) B8CSEL with k = 6-bit fragments.

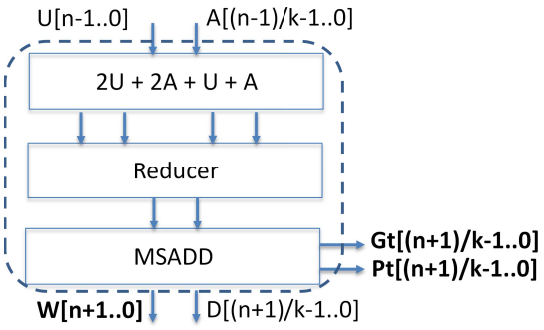

Fig. 8. Multispeculative Tripler (MST) architecture.

B. The Tripler Functional Unit

The structure of the module responsible for computing $3X$ in partial carry-save format is shown in detail in this subsection. Figure 8 depicts the architecture of the Multispeculative Tripler (MST), which is composed of three main stages, namely: operand generation, the reducer and the MSADD phases. The MST receives $X = U + A$ and computes $T = W + D$. First, it should be noted that X is an n-bit word, while T is an $(n+2)$-bit word. Second, U and W are the result vectors that will be driven to the OMSM-B8, while A and D are carry vectors, i.e. there is a bit every k-bits. Third, the MST produces the corresponding generate and propagate k-bit group signals that will also be driven to the OMSM-B8.

In order to perform the aforementioned calculation, we decompose $3X$ into 4 operands, which are straightforward to compute because they imply constant shifts at most. That is $3X = 3(U+A) = 2U+2A+U+A$. This fact is illustrated in Figures 9a and 9b. Figure 9a shows the X computation in an MSADD with $k = 3$, while Figure 9b displays how U, $2U$, A and $2A$ are arranged. As can be observed, $2A$ and A can be simply concatenated, as $k > 1$ in partial carry-save format. Thus, there are some positions where more than 2 bits need to be added, so an adder cannot directly be utilized. In order

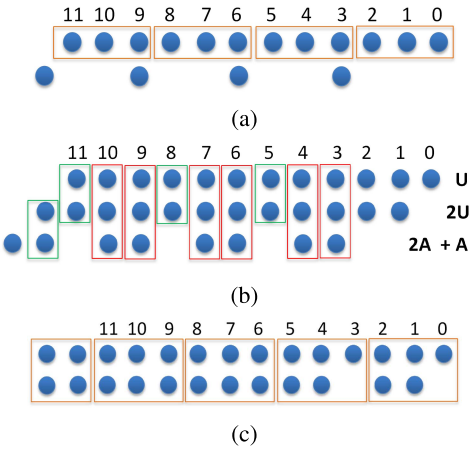

Fig. 9. Multispeculative Tripler stages for computing $3X$ with $n = 12$, $k = 3$. (a) $X = U + A$ calculation in partial carry-save. (b) Reducer stage for $3X = 2U + 2A + U + A$. (c) MSADD stage for the $3X$ calculation in partial carry-save.

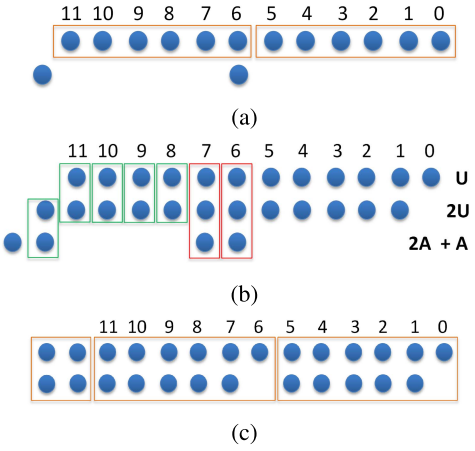

Fig. 10. Multispeculative Tripler stages for computing $3X$ with $n = 12$, $k = 6$. (a) $X = U + A$ calculation in partial carry-save. (b) Reducer stage for $3X = 2U + 2A + U + A$. (c) MSADD stage for the $3X$ calculation in partial carry-save.

to provide at most two bits in every position, a *reducer* block has been introduced. This block applies 3:2 (Full Adders) and 2:2 (Half Adders) counters in parallel to generate two bits at the most. Figure 9b shows where the 3:2 counters are applied with a red box, while the 2:2 counters are represented by green boxes. Finally, Figure 9c depicts how the reduced vectors are added with an MSADD employing $k = 3$. Figure 10 shows a similar example but utilizing $k = 6$. First, as can be observed, reductions are not necessary in the least significant fragment. Second, for the rest of the fragments, a 3:2 counter is required for the first two least significant positions, while half adders are employed for the rest of positions within the fragment.

Algorithm 1 synthesizes the ideas shown in both Figures 9 and 10, and describes a method to reduce the partial carry-save $3X = 2U + 2A + U + A$ value to just two vectors U', A'.

C. Proposed Method

After describing how to implement the required partial carry-save units, our proposed method for deploying partial carry-save datapaths is outlined in Algorithm 2. As can be observed, given a radix $R = 2^r$, a fragment size k and a

Algorithm 1 Reducer(n,k,U,A)

Input: 2U, U, 2A, A
Output: U′,A′ ▷ return the reduced values

$U' \leftarrow U, A' \leftarrow A$
$i \leftarrow 1$ ▷ Fragment 0 is not reduced
while $i < n/k$ **do**
  $3to2Counter(i * k, U', A')$
  $3to2Counter(i * k + 1, U', A')$
  $j \leftarrow 2$
  while $j < k$ **do**
    $2to2Counter(i * k + j, U', A')$
    $j \leftarrow j + 1$
  end while
  $i \leftarrow i + 1$
end while
$2to2Counter(n + 1, U', A')$

Algorithm 2 How to Deploy Partial Carry-Save Datapaths

Input: G, RS, R=2^r, k ▷ G is a DFG, RS is the resource set
Output: Partial carry-save datapath

$H \leftarrow hardMultiples(R)$
$MSU \leftarrow msUnits(H, k)$ ▷ Multispeculative units for H
$RS \leftarrow RS \cup MSU$
▷ High-level synthesis part
for each *product* $p \in G$ **do**
  for each *hard multiple* $h \in H$ **do**
    $InsertHardMultipleNode(h, p, G)$
  end for
end for
▷ Arithmetic part
for each *multiplier* $m \in RS$ **do**
  $CSEL(m.x, k)$ ▷ CSEL technique for the multiplicand
  for each *hard multiple* $h \in H$ **do**
    $CSEL(m.h, k)$ ▷ m.h is a hard multiple in partial carry-save
  end for
  $CSBE(m.y, k)$ ▷ CSBE recoding for the multiplier
end for
▷ High-level synthesis again
$ScheduleNBind(G, RS, k)$

resource set RS composed of partial carry-save units, first we generate the Multispeculative Units for the hard multiples, employing Algorithm 1 for the reducer stage. Second, it is necessary to insert as many nodes as the number of hard multiples (H), and for each multiplier it is necessary to apply the CSEL technique to any hard multiple received in partial carry-save format (as in the case of W in Figure 5). The last step consists of scheduling and binding the whole DFG.

VI. Illustrative Example

In this section a detailed example of use will illustrate how the multiplier presented in this paper works. First, Figure 11 depicts the conventional radix 8 Booth encoding for comparison purposes. Second, Figure 12 displays an

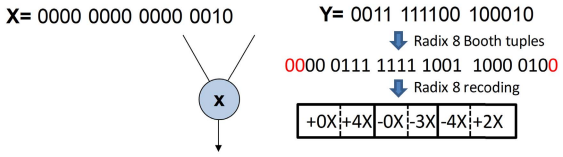

Fig. 11. A multiplier with non-redundant operands and radix 8 Booth encoding

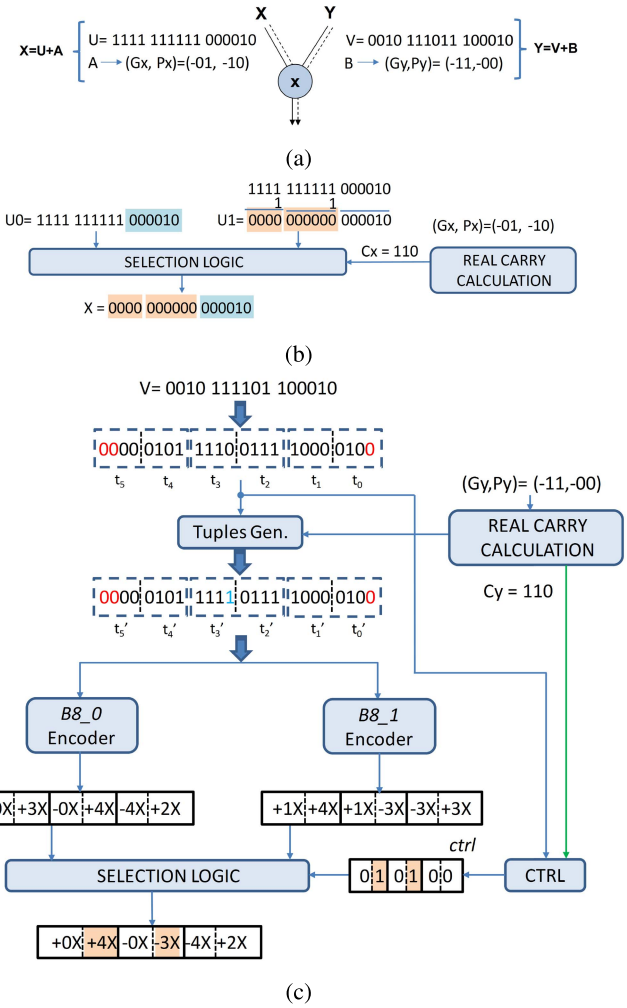

Fig. 12. Example of use of the On-the-Fly Correcting Radix 8 Booth Multispeculative Multiplier. (a) A multiplier with partial carry-save operands, $k = 6$. (b) Reconstructing the non-redundant multiplicand. (c) Multiplier branch: radix 8 carry select Booth encoding generates the multiples.

example of use based on the proposed OMSM-B8, where fragments of $k = 6$ bits are considered. Figure 12a displays how the fragments are driven to the multiplier.

Figure 12b depicts how the non-redundant form of the multiplicand operand is computed by applying the carry-select technique in the same fashion as in Figure 4b. Figure 12c displays how the B8CSBE works. First, the Y Real Calculation Unit finishes computing the actual carries and then all the fragments can start to recode. Second, the tuples t'_p are properly composed according to the cell depicted in Figure 6. Two encodings are generated in parallel (Booth_0 and Booth_1), and then the correct one is chosen for every tuple. The B8_0 encoder implements the conventional radix 8 recoding, while the B8_1 encoder implements the recoding specified by Table III. Finally, the correct recoding is selected thanks to the *ctrl* signals. As can be observed, the final encoding coincides with that provided by the conventional radix 8 Booth recoding shown in Figure 11.

Finally, it is interesting to note that as in the case of Figure 4c, in Figure 12c there are some bits in blue. These are least significant bits that have been modified because of the equations described in the appendix and deployed in the generic cell shown in Figure 6. This is the case of the least significant bit of tuple t'_3. The corresponding input tuple is $t_3 = v_{11}v_{10}v_9v_8 =$ "1110", but $t_2 = v_8v_7v_6v_5 =$ "0111" and according to Table III $msb_2 =$ '1' and $ctrl_2 =$ '1'. Thus, $lsb_3 = msb_2 =$ '1'.

VII. Experiments

In order to evaluate the features of the proposed OMSM-B8, several experiments have been carried out, in which each design is described at the gate level in VHDL and the functionalities of the OMSM-B8 units have been verified by ModelSim for randomly generated input patterns. The designs have been synthesized using a 65 nm TSMC standard-cell target-library as well as the Synopsys Design Compiler. In order to obtain the power results, the toggle rate was set to 0.1. First, the outcome of the multiplier syntheses will be presented. Second, the syntheses of several well-known benchmarks will be compared to demonstrate the improvements introduced by our approach.

A. Multiplier Synthesis

In this first experiment several multipliers with different bitwidths and fragment sizes have been synthesized. Different implementation styles have been applied too, namely: radix 4 Booth multiplier (*B4*), radix 8 Booth multiplier (*B8*), radix 4 OMSMs for $k = 4$ and $k = 8$ (*OMSM-B4-k4* and *OMSM-B4-k8*, respectively) and radix 8 OMSMs for $k = 3$, $k = 6$ and $k = 9$ (*OMSM-B8-3X-k3*, *OMSM-B8-3X-k6* and *OMSM-B8-3X-k9*, respectively). Furthermore, the library multiplication function and the combinational integer multipliers provided by a well-established tool as *Flopoco-v4.1.2* [44], [45] have been synthesized to complete the study. A timing constraint of 1 ns has been set, although for the largest designs it is violated. The results are shown in Tables IV, V, VI and VII.

As observed in Table IV, although all the proposals have a similar delay because of the timing constraint, the OMSM-B8-3X multipliers typically possess the lowest delay. On average, the OMSM-B8-3X exhibits around a 5% lower delay with respect to the B8 conventional multipliers (12% best case, for $n = 64$ and $k = 3$), a 4%-5% lower delay with respect to the B4 multipliers and a 6% average lower delay with respect to the OMSM-B4 with $k = 4$ and $k = 8$, respectively. Moreover, our radix 8-based OMSM scales better than the OMSM with radix 4. The library and the Flopoco multipliers are only comparable when $n = 8$-bits, for the rest of the cases they are unable to meet the timing constraint by a considerable amount of delay. On the other hand, the poor performance of the library function is due to the fact that typically synthesis tools do not work so well when the bitwidth is large [46].

TABLE IV

$n \times n$ MULTIPLIERS DELAY (ns). MAXIMUM DELAY CONSTRAINT: 1 ns

Bitwidth	B4	B8	OMSM-B4-k4 [26]	OMSM-B4-k8 [26]	OMSM-B8-3X-k3	OMSM-B8-3X-k6	OMSM-B8-3X-k9	Library	Flopoco
8x8	9.80E-01	9.60E-01	9.90E-01	-	1.00E+00	9.40E-01	-	9.30E-01	9.20E-01
16x16	1.00E+00	1.00E+00	1.02E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.26E+00	1.27E+00
32x32	1.11E+00	1.13E+00	1.13E+00	1.15E+00	1.05E+00	1.05E+00	1.04E+00	2.48E+00	2.41E+00
64x64	1.35E+00	1.40E+00	1.37E+00	1.41E+00	1.23E+00	1.27E+00	1.31E+00	4.90E+00	2.68E+00
128x128	1.73E+00	1.76E+00	1.72E+00	1.72E+00	1.56E+00	1.57E+00	1.59E+00	1.01E+01	3.33E+00

TABLE V

$n \times n$ MULTIPLIERS AREA (μm^2). MAXIMUM DELAY CONSTRAINT: 1 ns

Bitwidth	B4	B8	OMSM-B4-k4 [26]	OMSM-B4-k8 [26]	OMSM-B8-3X-k3	OMSM-B8-3X-k6	OMSM-B8-3X-k9	Library	Flopoco
8x8	1.9E+03	1.3E+03	2.1E+03	-	1.5E+03	1.4E+03	-	6.3E+02	1.5E+03
16x16	6.6E+03	5.7E+03	8.1E+03	8.0E+03	5.3E+03	5.4E+03	5.4E+03	3.7E+03	3.8E+03
32x32	2.3E+04	2.3E+04	2.8E+04	2.9E+04	2.1E+04	2.1E+04	2.2E+04	1.4E+04	1.8E+04
64x64	8.3E+04	8.5E+04	9.7E+04	9.8E+04	7.8E+04	7.7E+04	7.9E+04	5.3E+04	6.1E+04
128x128	3.1E+05	3.0E+05	3.5E+05	3.5E+05	2.8E+05	2.8E+05	2.8E+05	2.0E+05	2.39E+05

TABLE VI

$n \times n$ MULTIPLIERS POWER (μW). MAXIMUM DELAY CONSTRAINT: 1 ns

Bitwidth	B4	B8	OMSM-B4-k4 [26]	OMSM-B4-k8 [26]	OMSM-B8-3X-k3	OMSM-B8-3X-k6	OMSM-B8-3X-k9	Library	Flopoco
8x8	6.87E+02	5.86E+02	7.62E+02	-	7.56E+02	6.09E+02	-	3.06E+02	4.25E+02
16x16	2.99E+03	3.08E+03	4.07E+03	3.70E+03	3.36E+03	3.01E+03	2.90E+03	2.40E+03	2.37E+03
32x32	1.25E+04	1.38E+04	1.68E+04	1.56E+04	1.64E+04	1.40E+04	1.41E+04	1.21E+04	1.22E+04
64x64	4.76E+04	5.41E+04	6.18E+04	5.85E+04	6.05E+04	5.40E+04	5.26E+04	5.57E+04	5.13E+04
128x128	1.79E+05	1.91E+05	2.32E+05	2.15E+05	2.24E+05	1.99E+05	1.91E+05	2.32E+05	1.93E+05

TABLE VII

$n \times n$ MULTIPLIERS ENERGY (PJ). MAXIMUM DELAY CONSTRAINT: 1 ns

Bitwidth	B4	B8	OMSM-B4-k4 [26]	OMSM-B4-k8 [26]	OMSM-B8-3X-k3	OMSM-B8-3X-k6	OMSM-B8-3X-k9	Library	Flopoco
8x8	6.74E-01	5.63E-01	7.54E-01	-	7.56E-01	5.73E-01	-	2.85E-01	3.91E-01
16x16	2.99E+00	3.08E+00	4.15E+00	3.70E+00	3.36E+00	3.01E+00	2.90E+00	3.03E+00	3.00E+00
32x32	1.38E+01	1.56E+01	1.90E+01	1.80E+01	1.72E+01	1.47E+01	1.47E+01	3.01E+01	2.95E+01
64x64	6.43E+01	7.58E+01	8.46E+01	8.24E+01	7.44E+01	6.86E+01	6.90E+01	2.73E+02	1.38E+02
128x128	3.09E+02	3.36E+02	4.00E+02	3.70E+02	3.49E+02	3.12E+02	3.04E+02	2.34E+03	6.44E+02

In terms of area, as Table V shows, the OMSM-B8-3X multipliers take advantage of their radix 8-based implementation. Besides, as they possess a shorter critical path than B8 multipliers, their syntheses even require less area. On average, the OMSM-B8-3X multipliers are 5%-6% smaller than B8 multipliers, 14% smaller than B4 multipliers and around 26% smaller than the OMSM-B4 implementations, respectively. On the other hand, the library and the Flopoco multipliers provide smaller implementations.

Regarding the power, according to Table VI the OMSM-B8-3X-k3 multipliers suffer a certain penalty with respect to the B8 implementations (16.3%), but this becomes negligible for $k = 6$ and $k = 9$. In this tightly timing constrained scenario, the B4 multipliers possess the lowest power consumption but for $n = 8$. When compared with the OMSM-B4 multipliers, the OMSM-B8-3X ones require less power. It is interesting to note that for a certain fragment size, e.g. $k = 3$ or $k = 6$, the OMSM-B8-3X implementations are less power hungry than OMSM-B4 multipliers with larger fragment sizes, e.g $k = 4$ (6.5%) and $k = 8$ (11%), respectively. The library and the Flopoco multipliers show good behavior when the bitwidth is small, but power consumption becomes similar and then higher for $n \geq$ 32-bits.

Finally, in terms of energy, as observed in Table VII, the OMSM-B8-3X multipliers offer different tradeoffs, in general being more efficient as the fragment size increases. Specifically, with respect to the B8 implementation, there is a 9.5% energy increase for $k = 3$, but for $k = 6$ and $k = 9$ there are 4.8% and 6.7% decreases. The OMSM-B8-3X-k3 multipliers suffer 15% energy penalty with respect to the B4 multipliers, but with $k = 6$ the difference is negligible and with $k = 9$ there is even a 2% reduction. With respect to the OMSM-B4 implementations, the OMSM-B8-3X-k3 multipliers requires 11.5% less energy than the OMSM-B4-k4 ones, while the OMSM-B8-3X-k6 multipliers consume 16.7% less energy than the OMSM-B4-k8 multipliers. Overall, the OMSM-B8-3X-k9 multipliers are the most energy efficient implementation. The library and the Flopoco multipliers are better for $n = 8$-bits, similar for $n = 16$-bits and then consume more than twice (7.58X in the worst case) when compared with the baseline B4 multipliers.

B. Datapath Synthesis

In this second experiment, several 32-bit circuits (datapath+ controller) have been synthesized with several implementation styles, namely: radix 4 Booth multiplier (*B4*), radix 8 Booth multiplier without decoupling the $3X$ calculation (*B8*), and then the same multiplier but decoupling it (*B8-3X*). Furthermore, three partial-carry save radix 8 Booth-based

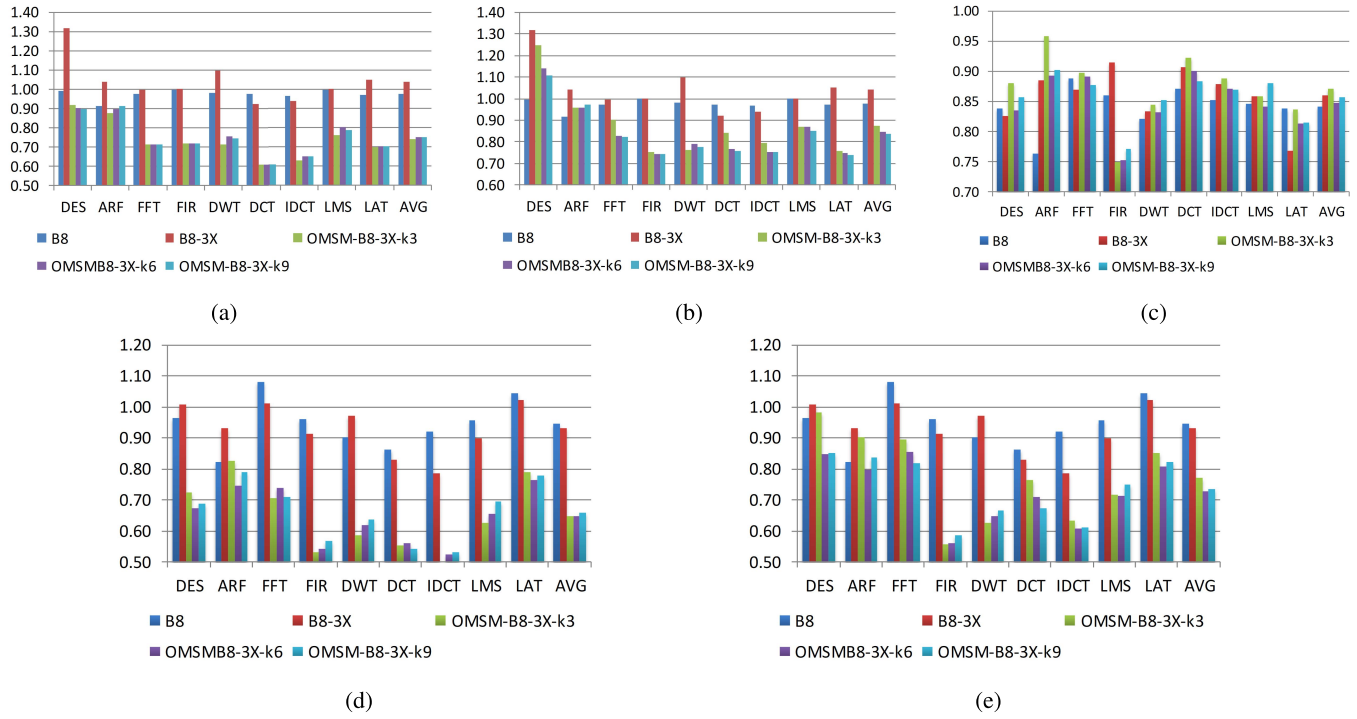

Fig. 13. Datapath synthesis results normalized with respect to the radix 4 Booth implementation. The resource set is composed of 2 adders, 2 multipliers and 2 triplers in the corresponding multispeculative or non-speculative form. (a) Execution time with modeled values. (b) Execution time with equiprobable values. (c) Area. (d) Energy with modeled values [47]. (e) Energy with equiprobable values.

TABLE VIII

32-BIT KOGGE-STONE ADDERS DELAY AND LATENCY. MAXIMUM DELAY CONSTRAINT: 1ns

Adder	Delay (ns)	Latency (cycles)
MSKS-k3	0.23	1
MSKS-k6	0.34	1
MSKS-k9	0.36	1
MSKS	0.75	2

TABLE IX

32 × 32 BIT MULTIPLIERS DELAY AND LATENCY. MAXIMUM DELAY CONSTRAINT: 1ns

Multiplier	Delay (ns)	Latency (cycles)
OMSM-B8-k3	1.05	3
OMSM-B8-k6	1.05	3
OMSM-B8-k9	1.04	3
B8-3X	1.03	3
B8	1.13	3
Booth4	1.11	3

implementations which decouple the $3X$ multiple have been considered too, with $k = 3$, $k = 6$ and $k = 9$ (labelled as *OMSM-B8-3X-k3*, *OMSM-B8-3X-k6* and *OMSM-B8-3X-k9*, respectively). The FU latencies are shown in Tables VIII and IX. These latencies have been calculated using the delay of the fastest adder ($MSKS - k3$, Multi-speculative Kogge-Stone with $k = 3$) as baseline (δ_{base}), and considering $\delta_{margin} = 0.3$ ns as design margin for the clock period [48]. Thus, if δ_{FU} is the delay of an FU, the latency is computed as $\lceil(\delta_{FU} + \delta_{margin})/(\delta_{base} + \delta_{margin})\rceil$. Then, as the largest latency is 3 cycles for the multipliers, the cycle time has been constrained to $(max(\delta_{FU}) + \delta_{margin})/3$ ns in this experiment. A similar list-based schedule and binding algorithm as in [25] has been considered. It should be noted that a *stallable* controller like the one presented in [10], [12], [49], and [50] has been deployed in these datapaths to generate the non-redundant form of the circuit outputs. Average latencies have been calculated for two scenarios, namely: by simulating the datapaths with modeled values and by considering equiprobability of bit occurrences [10], [47], aka. random inputs. Cycle time is given by the Design Compiler.

Figure 13 contains the execution time, area and energy results corresponding to the aforementioned benchmarks. The last column in the charts contains the average values. It should be noted that the results have been normalized with respect to the B4 values. As can be observed in Figures 13a and 13b, in this tightly time-constrained scenario decoupling the $3X$ multiple (B8-3X) is not enough for outperforming B4 or B8. Nevertheless, the OMSM-B8-3X implementations achieve average reductions of 25%-26% when modeling the values, and 12%-16% considering equiprobability of bits.

Area results are shown in Figure 13c. The smallest datapaths are those provided by the pure B8 implementation. Nevertheless, the rest of the implementations that decouple the $3X$ calculation achieve a lower average area than the baseline. This reduction is 14% for B8-3X, and ranges from 13% to 15% for the OMSM-B8-3X styles. Theoretically, and with softer timing constraints, the smaller the fragment, the bigger the area. When the fragments are smaller, the critical path is shorter but the number of generate and propagate signals to route and store is higher. Nevertheless, the tight timing constraint affects the OMSM-B8-3X-k9 implementations.

The energy consumption can be observed in Figures 13d and 13e. The radix 8 datapaths in general possess lower energy consumption than the baseline, OMSM-B8-3X being the most

energy efficient. The gains range between 34%-35% with modeled values, and 23%-27% considering random inputs. As shown, both the execution time and energy depend on the input data, but the OMSM-B8-3X implementations have behaved better than the baseline in both scenarios.

VIII. Conclusions

In this paper a partial carry-save radix 8 Booth-based multiplier has been presented. The starting point of this multiplier is a Dataflow Graph where the $3X$ hard multiple has been decoupled and inserted as an independent operation. Then, the radix 8 Booth recoder has been modified to efficiently encode the partial carry-save multiplier operand. Several propositions are included in the appendix that generalize this design for an arbitrary 2^r radix as well as a k-bit fragment, although in this work $r = 3$ has been employed. Furthermore, the units responsible for the $3X$ calculation have been deployed for the partial carry-save format, and the algorithm that is used for this purpose has been presented.

Our experiments have studied the tradeoffs obtained with several fragment sizes. From the synthesis results we can conclude that the On-The-Fly Correcting Radix 8 Booth Multispeculative Multipliers outperform their prior version with radix 4 [26] as they are more efficient in terms of energy in comparison with radix 4 and radix 8 implementations.

Regarding the formal basis presented in the appendix and given a 2^r radix Booth encoding, decoupling all the hard multiples and computing them in partial carry-save units as the Multispeculative Triplers would allow working with computation times depending on the $k-bit$ fragments. Hence, in the future the challenge is to apply a similar idea to larger radix values in order to study further promising tradeoffs.

Appendix
On the Necessity For Alignment and the Control Signals Of the Encoder

In this appendix, all the theorems and propositions that are the basis for the proposed Booth encoder are formulated and proved.

Theorem 1: If $R = 2^r$ and k-bit fragments are being considered, and $k \geq r$, all the tuples are aligned iff $k \ mod \ r = 0$.

Proof: First, we will prove that if all the tuples are aligned, then $k \ mod \ r = 0$. By definition of *tuples alignment*, each pair of radix $R = 2^r$ tuples $t_p = v_{p*r+r-1}v_{p*r+r-2}\ldots v_{p*r}v_{p*r-1}$ and $t_q = v_{q*r+r-1}v_{q*r+r-2}\ldots v_{q*r}v_{q*r-1}$ will receive a carry-in signal in positions i and j, respectively, such that $i_r \equiv j_r$. That is, for every r bits there is a carry-in. These carry-in signals can be either real carries or internal carries, but in our scenario at least there must be one real carry proceeding from a Carry Calculation Unit. Hence, let us suppose that v_f is the first bit receiving a real carry-in besides v_0, which receives $Cy_0 = '0'$. If v_f receives Cy_1, then the beginning of a new fragment is located at the f^{th} position. So the previous fragment started at position 0 and ended at position $f-1 = m*r-1$, for a certain $m > 0$. As all the fragments possess the same length, we can conclude that $k = m*r$, or $k \ mod \ r = 0$.

Second, let us prove that if $k \ mod \ r = 0$, then all the tuples are aligned. If $k \ mod \ r = 0$ then $k = m * r$ for a certain $m > 0$. Because of the $Booth\_1$ encoding, any tuple $t_p = v_{p*r+r-1}v_{p*r+r-2}\ldots v_{p*r}v_{p*r-1}$ receives a carry-in with a weight $p * r$ and produces a carry-out with a weight $p * r + r$, which will affect the following tuple $t_{p+1} = v_{(p+1)*r+r-1}v_{(p+1)*r+r-2}\ldots v_{(p+1)*r}v_{(p+1)*r-1}$. That is, t_p and t_{p+1} are aligned because they receive a carry-in in positions $[p * r]_r \equiv [(p+1) * r]_r \equiv 0_r$. The only possibility of unalignment may happen with real carries, when there can exist an extra carry affecting a different bit. However, if $k = m * r$, $Cy_0 = '0'$ must be added to v_0, Cy_1 to v_k, Cy_2 to v_{2*k} and in general Cy_i will be added to $v_{i*k} = v_{i*m*r}$, so $(i * m * r) \ mod \ r = 0$ and then all the tuples will be aligned. □

Corollary 1: Given radix $R = 2^3 = 8$, using the fragment size $k = 3 * m$ ensures all the fragments are aligned.

Proof: The proof is a follow-up of Theorem 1, using $r = 3$. □

Corollary 1 establishes the conditions in which the Booth tuples are aligned in radix 8. If this happens, it is possible to design an encoder based on a cell with a similar structure to that used in [26], whose bitwidths have been adapted to radix 8 in Figure 6. Table III shows the truth table corresponding to the $B8\_1$ recoder. It should be noted that the sign ($sign_p$) and most significant bit (msb_p) columns have been fused, as they possess the same values.

Finally, by using Theorem 1 it is possible to extend the the rest of the equations defined [26] for radix 4, and which manage the cell depicted in Figure 6. Firstly, Proposition 1 establishes how many Booth tuples belong to every fragment.

Proposition 1: If the radix 2^r Booth tuples are aligned, every k-bit fragment contains exactly k/r tuples.

Proof: If the tuples are aligned, $k \ mod \ r = 0$, with $k \geq r$. In other words, $k = w * r$, for a certain w. In base 2^r, each tuple $t_p = v_{p*r+r-1}v_{p*r+r-2}\ldots v_{p*r}v_{p*r-1}$ is composed of $r+1$ bits, the most significant bit of tuple t_p overlapping with the least significant bit of tuple t_{p+1}. Hence, it can be considered that every tuple *covers* r different bits. Therefore, it is clear that there will be exactly $w = k/r$ tuples within every fragment. □

Proposition 2 defines the position of the first Booth tuple within every k-bit fragment and for every radix 2^r.

Proposition 2: If t_p is the first aligned 2^r Booth tuple belonging to the k-bit fragment i, then $p = i * (k/r)$.

Proof: Let us prove this using the induction principle over i. In fragment 0, the first tuple is t_0, so the base case is true. Provided that in fragment i the first tuple is t_p with $p = i * (k/r)$ and that the tuples are aligned, there will be exactly k/r tuples within fragment i, $t_{p+k/r-1}$ being the last one. Hence, the first tuple belonging to fragment $i+1$ is $t_{p+k/r}$, with $p = i * (k/r) + k/r = (i+1) * (k/r)$. □

Proposition 3 defines the control signal $ctrl_p$ that appears in Figure 6 and whose objective is to decide which encoding must be employed for tuple t_p.

Proposition 3: Let us consider an aligned radix 2^r Booth tuple t_p belonging to the k-bit fragment i. In addition, provided that fragment i receives the real carry Cy_i as its carry-in,

and every tuple t_p produces an internal carry-out c_p, the control signal $ctrl_p$ that determines which encoding, Booth_0 ($b0$) or Booth_1 ($b1$), is used to generate the multiples is defined as follows:

$$ctrl_p = \begin{cases} Cy_i, & if\ \ p\ mod\ k = 0, \\ Cy_i \cdot \bigwedge_{m=i*(k/r)}^{p-1} (c_m) & otherwise\ . \end{cases} \quad (2)$$

Proof: There are two cases: first, if $p\ mod\ (k/r) = 0$, then $p = i * (k/r)$. This means that p is a position corresponding to the least significant bit of fragment i. Hence, the carry-in proceeds from the Y carry calculation unit. In this case, the $b1$ encoding must be selected when Cy_i is '1', and $b0$ otherwise. Thereby, Cy_i alone is enough to decide this.

If $p\ mod\ k \neq 0$, the actual carry-in to t_p is '1' iff the carry-in of the fragment is '1', i.e. $Cy_i = '1'$, and all the previous internal carry-out signals inside the fragment are '1', that is, those generated by the tuples $t_{p-1}, t_{p-2}, \ldots, t_{i*k/r}$, i.e. iff $c_{p-1}, c_{p-2}, \ldots c_{i*(k/r)}$ are all '1'. If any of these internal carry-out signals is '0', i.e. t_{p-j}, with $j > 0$, this means that t_{p-j+1} receives a '0' carry-in. Thus, t_{p-j+1} must select the $b0$ encoding. As the $b0$ encoding produces no carry-out, the following tuples $t_{p-j+2}, t_{p-j+3}, \ldots, t_{p-1}, t_p$ must choose the $b0$ encoding. □

According to Proposition 3, when $p\ mod\ (k/r) \neq 0$, the $ctrl_p$ signal depends on several carries. Nevertheless, first these carries only depend on the input bits and can be calculated in parallel and, second, the purpose of this work is to consider small values of k, so the number of internal carries is quite small. In practice, there will never be more than 3 tuples within the fragment, which will be the maximum number of inputs for the logic AND generating $ctrl_p$.

Finally, Proposition 4 defines the least significant bit of a Booth tuple t_p, depending on the Booth encoding employed for the prior tuple t_{p-1}.

Proposition 4: Given two consecutive aligned 2^r Booth tuples $t_p = v_{p*r+r-1}v_{p*r+r-2}\ldots v_{p*r}v_{p*r-1}$ and $t_{p-1} = v_{(p-1)*r+r-1}v_{(p-1)*r+r-2}\ldots v_{(p-1)*r}v_{(p-1)*r-1}$, the actual Booth tuple $t'_p = v_{p*r+r-1}v_{p*r+r-2}\ldots v_{p*r}lsb_p$ that will be driven to the Booth encoders, and provided that tuple t_j produces $ctrl_j$ and msb_j as control and msb signals, respectively, the new bit lsb_p is defined as follows:

$$lsb_p = msb_{p-1} \cdot ctrl_{p-1} + v_{p*r-1} \cdot \overline{ctrl_{p-1}}. \quad (3)$$

Proof: The $ctrl_{p-1}$ signal establishes that there exists a chain of internal carry-out signals within the fragment. Hence, if $ctrl_{p-1}$, the Booth_1 encoding must be selected for t_{p-1}. Thus, t_{p-1} may produce a new msb when $ctrl_{p-1} = '1'$. In this case, msb_{p-1} must be selected as lsb_p.

On the other hand, whenever $ctrl_{p-1}$ does not hold, such a chain of internal carry-out signals does not exist, and then the multiplier bit v_{p*r-1} must remain as lsb_p. □

These four propositions have been defined for a generic 2^r radix and k-bit fragments. In this paper $r = 3$ has been used.

REFERENCES

[1] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. New York, NY, USA: McGraw-Hill, 1994.

[2] S. Gupta, A. Nicolau, N. D. Dutt, and R. K. Gupta, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Norwell, MA, USA: Kluwer, 2004.

[3] P. P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed. Dordrecht, The Netherlands: Springer, 2008. [Online]. Available: https://www.springer.com/gb/book/9781402085871

[4] D. De Caro, E. Napoli, D. Esposito, G. Castellano, N. Petra, and A. G. M. Strollo, "Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 5, pp. 1187–1200, May 2017.

[5] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[6] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, "An algorithmic approach for generic parallel adders," in *Proc. Int. Conf. Comput. Aided Design (ICCAD)*, 2003, pp. 734–740.

[7] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, Mar. 2004.

[8] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *Proc. Design, Automat. Test Eur. (DATE)*, Mar. 2008, pp. 1250–1255.

[9] A. Cilardo, "A new speculative addition architecture suitable for two's complement operations," in *Proc. Design, Automat. Test Eur. (DATE)*, Apr. 2009, pp. 664–669.

[10] A. A. Del Barrio, R. Hermida, and S. O. Memik, "Exploring the energy efficiency of multispeculative adders," in *Proc. Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 309–315.

[11] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1761–1774, Oct. 2008.

[12] A. A. Del Barrio, R. Hermida, S. O. Memik, J. M. Mendias, and M. C. Molina, "Improving circuit performance with multispeculative additive trees in high-level synthesis," *Microelectron. J.*, vol. 45, no. 11, pp. 1470–1479, Nov. 2014.

[13] P.-M. Seidel, L. D. McFearin, and D. W. Matula, "Binary multiplication radix-32 and radix-256," in *Proc. IEEE Symp. Comput. Arithmetic (ARITH)*, Jun. 2001, pp. 23–32.

[14] M. Ercegovac and T. Lang, *Digital Arithmetic*, 1st ed. San Mateo, CA, USA: Morgan Kaufmann, 2003.

[15] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. Natick, MA, USA: AK Peters, 2002.

[16] P. M. Seidel, L. D. McFearin, and D. W. Matula, "Secondary radix recodings for higher radix multipliers," *IEEE Trans. Comput.*, vol. 54, no. 2, pp. 111–123, Feb. 2005.

[17] E. J. King and E. E. Swartzlander, "Data-dependent truncation scheme for parallel multipliers," in *Proc. Conf. Rec. 31st Asilomar Conf. Signals, Syst. Comput.*, vol. 2, Nov. 1997, pp. 1178–1182.

[18] H. J. Ko and S. F. Hsiao, "Design and application of faithfully rounded and truncated multipliers with combined deletion, reduction, truncation, and rounding," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 58, no. 5, pp. 304–308, May 2011.

[19] T. A. Drane, T. M. Rose, and G. A. Constantinides, "On the systematic creation of faithfully rounded truncated multipliers and arrays," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2513–2525, Oct. 2014.

[20] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, Jan. 1951.

[21] J.-Y. Kang and J. L. Gaudiot, "A simple high-speed multiplier design," *IEEE Trans. Comput.*, vol. 55, no. 10, pp. 1253–1258, Oct. 2006.

[22] S. R. Kuang, J. P. Wang, and C. Y. Guo, "Modified Booth multipliers with a regular partial product array," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 56, no. 5, pp. 404–408, May 2009.

[23] H. H. Saleh, B. S. Mohammad, and E. E. Swartzlander, "The optimum Booth radix for low power integer multipliers," in *Proc. 8th Int. IEEE Design Test Symp. (IDT)*, Dec. 2013, pp. 1–4.

[24] E. M. Schwarz, R. M. Averill, and L. J. Sigal, "A radix-8 CMOS S/390 multiplier," in *Proc. IEEE Symp. Comput. Arithmetic (ARITH)*, Jul. 1997, pp. 2–9.

[25] A. A. Del Barrio and R. Hermida, "A slack-based approach to efficiently deploy radix 8 booth multipliers," in *Proc. Design, Automat. Test Eur. (DATE)*, 2017, pp. 1153–1158.

[26] A. A. Del Barrio, R. Hermida, S. O. Memik, "A partial carry-save on-the-fly correction multispeculative multiplier," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3251–3264, Nov. 2016.

[27] B. S. Cherkauer and E. G. Friedman, "A hybrid radix-4/radix-8 low power signed multiplier architecture," *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, vol. 44, no. 8, pp. 656–659, Aug. 1997.
[28] J. Clouser *et al.*, "A 600-MHz superscalar floating-point processor," *IEEE J. Solid-State Circuits*, vol. 34, no. 7, pp. 1026–1029, Jul. 1999.
[29] J. A. Hidalgo-López, V. Moreno-Vergara, Ó. Oballe-Peinado, A. Daza, M. J. Martín-Vázquez, and A. Gago, "A radix-8 multiplier unit design for specific purpose," in *Proc. 13th Conf. Design Circuits Integr. Syst.*, vol. 10, 1998, pp. 1535–1546.
[30] G. D. Licciardo, C. Cappetta, L. Di Benedetto, and M. Vigliar, "Weighted partitioning for fast multiplierless multiple-constant convolution circuit," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 64, no. 1, pp. 66–70, Jan. 2017.
[31] I. Hatai, I. Chakrabarti, and S. Banerjee, "A computationally efficient reconfigurable constant multiplication architecture based on CSD decoded Vertical–Horizontal common sub-expression elimination algorithm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 1, pp. 130–140, Jan. 2018.
[32] N. Besli and R. G. Deshmukh, "A novel redundant binary signed-digit (RBSD) Booth's encoding," in *Proc. SoutheastCon*, Apr. 2002, pp. 426–431.
[33] H. A. H. Fahmy, A. A. Liddicoat, and M. J. Flynn, "Improving the effectiveness of floating point arithmetic," in *Proc. Asilomar Conf. Signals, Syst. Comput.*, Nov. 2001, pp. 875–879.
[34] H. A. H. Fahmy and M. J. Flynn, "The case for a redundant format in floating point arithmetic," in *Proc. Symp. Comput. Arithmetic (ARITH)*, Jun. 2003, pp. 95–102.
[35] A. Cilardo *et al.*, "High speed speculative multipliers based on speculative carry-save tree," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 12, pp. 3426–3435, Dec. 2014.
[36] H. Jiang, J. Han, F. Qiao, and F. Lombardi, "Approximate radix-8 booth multipliers for low-power and high-performance operation," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2638–2644, Aug. 2016.
[37] M. I. Ferguson and M. D. Ercegovac, "A multiplier with redundant operands," in *Proc. Asilomar Conf. Signals, Syst., Comput.*, Oct. 1999, pp. 1322–1326.
[38] S. Belloeil-Dupuis, R. Chotin-Avot, and H. Mehrez, "Exploring redundant arithmetics in computer-aided design of arithmetic datapaths," *Integr., VLSI J.*, vol. 46, pp. 104–118, Mar. 2013.
[39] G. W. Bewick, "Fast Multiplication: Algorithms and Implementation," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 1994.
[40] M. Daumas and D. W. Matula, "A Booth multiplier accepting both a redundant or a non redundant input with no additional delay," in *Proc. Int. Conf. Appl.-Specific Syst., Archit., Process.*, Jul. 2000, pp. 205–214.
[41] K. Tsoumanis, S. Xydis, C. Efstathiou, N. Moschopoulos, and K. Pekmestzi, "An optimized modified Booth recoder for efficient design of the add-multiply operator," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 4, pp. 1133–1143, Apr. 2014.
[42] Y. Dumonteix and H. Mehrez, "A family of redundant multipliers dedicated to fast computation for signal processing," in *Proc. Int. Symp. Circuits Syst. (ISCAS)*, May 2000, pp. 325–328.
[43] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida, "Ultra-low-power adder stage design for exascale floating point units," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3s, pp. 105-1–105-24, Mar. 2014.
[44] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design Test Comput.*, vol. 28, no. 4, pp. 18–27, Jul./Aug. 2011.
[45] F. de Dinechin, "Multiplication by rational constants," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 59, no. 2, pp. 98–102, Feb. 2012.
[46] G. Quan, J. P. Davis, S. Devarkal, and D. A. Buell, "High-level synthesis for large bit-width multipliers on FPGAs: A case study," in *Proc. 3rd IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2005, pp. 213–218.
[47] A. A. Del Barrio, R. Hermida, S. O. Memik, J. M. Mendias, and M. C. Molina, "Multispeculative addition applied to datapath synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 12, pp. 1817–1830, Dec. 2012.
[48] S. Lee and K. Choi, "High-level synthesis with distributed controller for fast timing closure," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2011, pp. 193–199.
[49] A. A. Del Barrio, R. Hermida, S. O. Memik, J. M. Mendias, and M. C. Molina, "Multispeculative additive trees in high-level synthesis," in *Proc. Design, Automat. Test Eur. (DATE)*, Mar. 2013, pp. 188–193.
[50] J. Cong, A. Liu, and B. Liu, "A variation-tolerant scheduler for better than worst-case behavioral synthesis," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2009, pp. 221–228.

Alberto A. Del Barrio (M'13) received the Ph.D. degree in computer science from the Complutense University of Madrid (UCM), Madrid, Spain, in 2011. Since 2017, he has been an Interim Associate Professor of computer science with the Department of Computer Architecture and System Engineering, UCM. His current research interests include arithmetic, design automation, video processing, and cognitive computing. He has served as a Reviewer for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and also for DAC, DATE, ICCAD, and CODES+ISSS conferences. He has also served as a Technical Program Committee Member on several conferences such as DATE and IEEE/IFIP EUC.

Román Hermida (SM'04) received the Ph.D. degree from the Complutense University of Madrid (UCM), Spain, in 1984. He is currently a Professor with the Department of Computer Architecture and Systems Engineering, UCM. His current research interests include design automation, computer architecture, reconfigurable computing, and embedded systems. He has been actively involved in program committees of several international conferences. He received the 2002 IEEE VLSI Transactions Best Paper Award. He served as the Program Chair and the General Chair for the International Symposium on System Synthesis in 2000 and 2001, respectively. He has also served as the Track Chair for CODES+ISSS and DATE.

Seda Ogrenci-Memik (SM'05) received the B.S. degree in electrical and electronic engineering from Bogazii University, Istanbul, Turkey, and the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles, CA, USA. She is currently an Associate Professor with the Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL, USA. Her research interests include embedded and reconfigurable computing, HLS, thermal-aware design automation, and thermal management for microprocessor systems. She has served as a technical program committee member, an organizing committee member, and the track chair of several conferences, including ICCAD, DATE, FPL, GLSVLSI, and ISVLSI. She received the National Science Foundation Early Career Development (CAREER) Award in 2006. She is currently serving on the Editorial Board of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION.