Multiplier Architecture with a Carry-Based Partial Product Encoding

Martin Langhammer
Intel Corporation, UK
martin.langhammer@intel.com

Bogdan Pasca
Intel Corporation, France
bogdan.pasca@intel.com

Igor Kucherenko
Intel Corporation, US
igor.kucherenko@intel.com

***Abstract*—Multipliers have always been an important component of computer architecture, but the increasing relevance of Artificial Intelligence (AI) has brought about a massive increase in the number of multipliers on all compute platforms. At the same time, multiplier use for signal processing has also increased unabated. Multiplier architectures have not changed appreciably over the recent past. In this paper, we introduce a new technique for calculating partial products, which can be used with known compression tree and adder combinations. We demonstrate the efficiency of our new multiplier by reporting results from 800MHz to 2GHz in a current 7nm production library, and comparing to the well-known modified Booth's radix 4 and radix 8 architectures.**

***Index Terms*—multiplier, partial-product, encoding, carry-chain**

I. Introduction

AI requires many different precisions and performance levels of multipliers. Low-precision multipliers (typically INT8 or smaller) are typically used for inference. Higher precision multipliers - often floating point, where a considerable portion of the area is the mantissa multiplier of low (UINT8 for bfloat16) or medium (UINT24 for FP32) precision - are used for training. Traditional signal processing applications are also increasing in functional density, and require higher numbers of medium precision (INT16 or INT24) multipliers. All of this motivates us to find the most efficient integer multiplier architecture possible.

In the last 30 years since Bewick [1], there have been few new multiplier architectures (which we define by the partial product generation methods), and many implementations are still dependent on a modified version [2] of Booth's algorithm [3]. (In this paper, any following references to Booth's algorithm will be to the modified form.) An alternate higher radix scheme [4] was developed for radices 32 and 256, using a compound of two smaller radices to reduce the number of additions in the partial product. Despite only using a single addition, the partial product generation is very deep, and may be prohibitively expensive for smaller multipliers.

Research has not stopped into other areas of multiplier use architectures: there are many soft-logic FPGA multipliers published [5], [6], [7], including very high precision cases [8], [9]. Decimal (rather than binary form) multipliers [10], [11] for ASIC have also been studied. The combination of many multipliers into dot products [12] is important for the construction of the vector-vector kernels so common in AI.

Some recent works are useful for multiplier optimization aside from the partial products, which we can use in conjunction with our new methods. Power optimization methods [13] and compression optimization [14] have been reported. There are also recent results for carry propagate adders [15], [16], [17]. These can be used at the output stage, but can also be used during partial product generations, *e.g.* to calculate the $2A + A$ value for the Booth's radix 8 partial products.

All of the higher order (greater than radix 2) schemes use multiples of the multiplicand operand. In contrast, our presented method will use properties of the multiplier operand.

We make two main contributions in this work:

- A new multiplier partial-product encoding scheme, compatible with standard reduction (compression) methods, which can be effectively used on even low-precision multipliers.
- We describe future work, which can take advantage of the reorganization of the delays of our proposed method, to more effectively implement higher radix multipliers.

II. Acknowledgements

After the review phase it was pointed out to us that some elements of this work already exist in the following US patent (https://patents.google.com/patent/US10466968B1/en). Our work was done independently of this patent without any prior knowledge of its existence.

III. Implementation

Most multiplier architectures are currently based on some variant of Booth's encoding, which creates partial products that are completely independent of each other. This allows for many more degrees of freedom in synthesis and placement, and is a major contributor to the high-performance multiplier-based systems in use today.

An example of how Booth's radix 4 encoding operates is depicted in Figure 1 for an 8-bit multiplier. Two

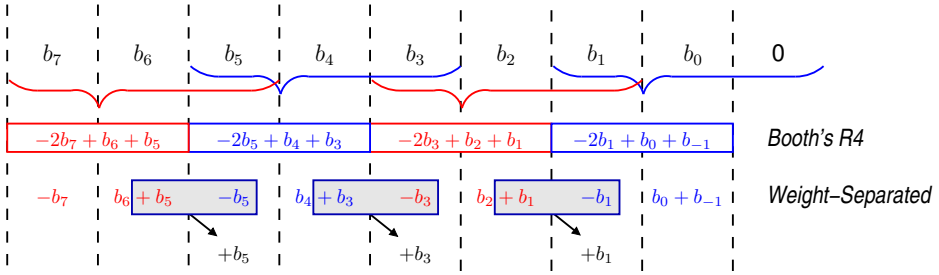

Fig. 1: Radix-4 Modified Booth Encoding for an 8-bit input. Weight-Separated encoded terms shown on the bottom of the figure, highlighting the redundancy aspect of the encoding.

TABLE I: Booth's Radix 4 Encoder

b_{2j+1} **CO**	b_{2j}	b_{2j-1} **CI**	**B** $b_{2j+1:2j}$	**M**	$B + b_{2j-1}$	**CO**
0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	0	1	1	1	0
0	1	1	1	2	2	0
1	0	0	2	-2	2	1
1	0	1	2	-1	3	1
1	1	0	3	-1	3	1
1	1	1	3	0	4	1

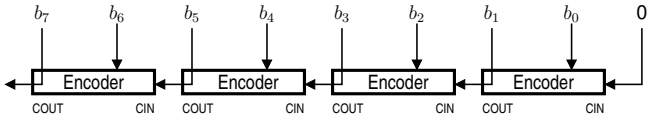

Fig. 2: Modified Booth's Radix-4 Chained Encoders represented as modules with carry-in and carry-out.

bits are encoded at a time, using a window of 3 bits from the input. Successive 3-bit windows have a 1-bit overlap. All dibits may be encoded in parallel as there are no direct dependencies between encoded dibits. The information carry-over between dibits is avoided by using the overlapping 3-bit windows. Nonetheless, the set of encoders may be depicted as a carry-chain structure, where the carry-out and carry-in of adjacent encoders correspond to the same input bit – as shown in Figure 2. Table I shows Booth's radix 4 encoding and Eq. 1 explicitly states the encoding equation.

$$B_j = -2b_{2j+1} + b_{2j} + b_{2j-1} \tag{1}$$

The M reported in Table I is the value for generating a partial product between encoded dibit B_j and the multiplicand A. Operations on signed operands encoded in 2's complement are generally implemented using a 1's complement (inversion), followed by the addition of the sign bit back into the vector to complete the 2's complement representation. The addition of the sign bit will require a carry operation, so all of the sign bits are combined into a single vector, and compressed with the rest of the partial products.

In one way, Booth's encoding operates as a CSD (canonic signed digit) arithmetic system. A run of '1' bits can be replaced by a single '1' at the first '0' after the MSB of the run, and a subtraction of a '1' at the LSB of the run.

Our new method can be understood by restating the Booth's radix 4 codes in a CSD-like form, which is shown in Table II. The CI is the MSB of the previous dibit, B is the current dibit, which can have values "00", "01", "10", and "11" while CO is the carry out, which is the MSB of the current dibit into the next dibit, or can be alternately seen as the '1' after the run of '1's in the current dibit. The operation on the multiplicand to make the partial product is denoted by M. For simplicity, and to avoid carries across the different encoders, "2" is treated as a run of '1's. To make the coding correct the operation for 2 is a -2x, which is the CO*2^2-2. In other words, the operation is x(4-2) = x2, which is what we want here.

Now we will do something counter-intuitive, and modify the encoding such that the multiplication by 2 is directly encoded as a x2 operation, as opposed to a x(4-2) operation. In other words, rather than coding the CO as the MSB of the B value (which will enter as a carry-in into the next dibit with weight 4), we will code the B as a x2, but only of the CI = 0 and in this case we will force the CO = 0. This means that encoding the current dibit will now depend not only on the overlapping 3-bit multiplier windows bits, but on the previous CIs as well.

But now we need a carry across the encoders again. Fortunately, this turns out to be inexpensive. We will define this carry-chain differently, because it will be across the dibit, rather than down a column. A CO is defined if there is either a generate or a propagate across the dibit. A generate occurs if there is a carry in and the dibit is 2 or greater, and a propagate occurs if the dibit is 3. The simplified generate and propagate equations (Eq. 2 and Eq. 3) used in our method are identical to the known equations used for carry-propagate adders.

$$p_i = b_{2j+1} \tag{2}$$

$$g_i = b_{2j+1} \cdot b_{2j} \tag{3}$$

TABLE II: Alternate form of the Booth's 4 Encoding. It replaces the "-2" entry with "+2", and modifies the carry-out from the encoder into the next encoder.

CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**+2**	**0**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

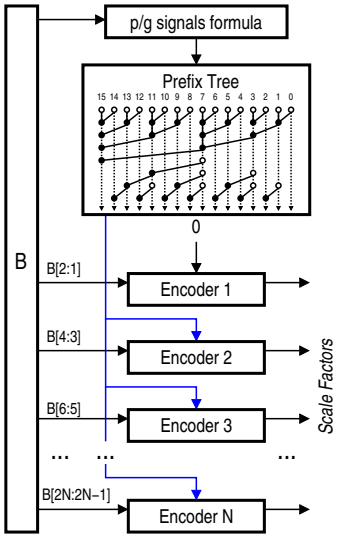

Fig. 3: Applying a Carry Chain to the Encoders.

We can use a prefix tree, or a CLA, to calculate this across all encoders, similar to the way a normal CPA is implemented, although here we have a different (horizontal vs. vertical) definition of the generate and propagate conditions. An implementation is shown in Figure 3. We call our method B4G3 (Booth's 4, Generate 3), for the form of Booth's radix 4 CSD structure, modified with a carry condition when the dibit is 3.

A. Encoder Structure

Figures 4 and 5 show the new partial product generator structure. The encoder, depicted in Figure 4, is only instantiated once per partial product. There are only two multiplier operand bits used (b_{2j} and b_{2j+1}) - the CIN input is the output of the carry chain, *i.e.* the prefix structure of Figure 3.

The partial product mux depicted in Figure 5 is applied at every bit position of each partial product. Note that the '$-2A$' value no longer needs to be considered, saving an AND gate and reducing the number of inputs to the OR gate by one (from four to three).

B. Analysis - Why this Works

At first glance, this approach adds complexity and significant logic depth to the Booth's radix 4 structure,

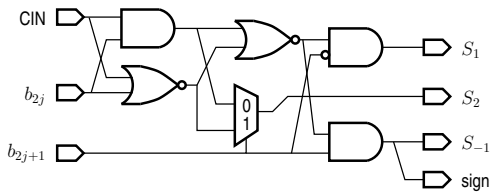

Fig. 4: B4G3 Encoder

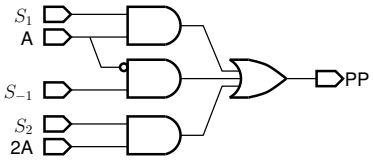

Fig. 5: B4G3 Partial Product Mux

while removing only a single mux input in the partial product encoding. But if we look at the area impact, we can see that the number of gates is significantly reduced. The representative prefix tree shown in Figure 3 has 16 inputs, which is for a relatively large example (multiplier operand precision of 32-bits). A 16-bit multiplier would have less than half this complexity, with 11 dots in the Brent-Kung prefix tree. The number of gates in this structure is approximately 33 individual gates. Removing a mux input (16-bits) from the 8 partial products would be about 256 individual gates (2 gates$\times$16 bits$\times$8 vectors) a significant difference which can be used by the synthesis tool to make a measurable improvement in the area and/or speed of the multiplier.

ASIC libraries contain components implementing compound gates, so the area saving with vary from process node to process node, but our proposed method will provide a useful resource reduction.

IV. Results

Figure 6 shows the R4G3 multiplier architecture results (for an example 12x12 signed multiplier) compared to the Synopsys Designware [18] multipliers for both Booth's radix 4 and radix 8 encoding. Our R4G3 multiplier uses the DesignWare compressor [19] and final CPA cores. The vertical axis shows the reported area (μm^2) in an Intel 7nm production library. All the reported areas are for synthesized, not placed and routed, results. The graph shows that the proposed multiplier outperforms both Synopsys multipliers for frequency goals ranging from 800MHz to 2.1GHz. Above this threshold the Synopsys Booth's radix 4 (B4) shows better results.

V. Future Work

We can also combine both our new method and known methods into a single structure, as shown in Figure 7. Our described method introduces an additional delay in the multiplier operand side (via the carry network, which we are implementing with a prefix tree in our example). Nonetheless, the overall number of gates in

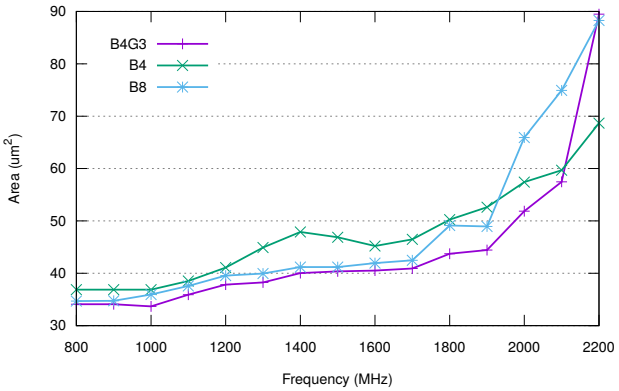

Fig. 6: Area comparison of R4G3, Booth's R4, Booth's R8 - Swept from 800MHz to 2.2 GHz - 12x12 Signed Case.

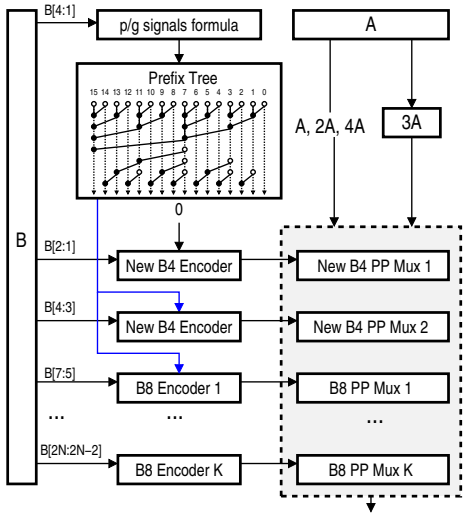

Fig. 7: Mixed Radix

the partial product generation is reduced by a sizeable amount.

Higher radix partial-product methods (such as Booth's radix 8) instead introduce a delay in the multiplicand operand side (shown in Figure 7 by the $3A$ value, which also requires a carry-chain implementation) but can reduce the number of gates and overall latency by reducing the number of partial products. Using a mix of the two methods may produce even better results.

VI. Conclusions

With the increasing densities of multipliers, efficient architectures are needed more than ever. Although research in arithmetic structures (such as compressors and adders) is still yielding results, little progress in partial product generation seems to have occurred recently.

We have described a new approach to multiplier partial-product generation that uses a carry-propagate structure to select partial product values. Previous high-radix approaches introduced a carry-propagate calculation to the multiplicand operand. Our proposed method instead uses the carry-propagate on the multiplier operand. We have shown that this significantly reduces the gate count in the partial-product generation, which translates to a smaller, faster multiplier implementation.

Finally, we have proposed follow-on experiments, where the carry-based encoding can be combined with higher radix (*e.g.* radix 8) partial-product generation.

References

[1] G. W. Bewick, "Fast multiplication: Algorithms and implementation," Ph.D. dissertation, Stanford University, Feb. 1994.

[2] O. L. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, Jan 1961.

[3] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951. [Online]. Available: +http://dx.doi.org/10.1093/qjmam/4.2.236

[4] P.-M. Seidel, L. McFearin, and D. Matula, "Binary multiplication radix-32 and radix-256," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 23–32.

[5] M. Langhammer and G. Baeckler, "High density and performance multiplication for FPGA," in *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, 2018, pp. 5–12. [Online]. Available: https://doi.org/10.1109/ARITH.2018.8464695

[6] E. G. Walters, "Partial-product generation and addition for multiplication in FPGAs with 6-input LUTs," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 1247–1251.

[7] M. Kumm, S. Abbas, and P. Zipf, "An efficient softcore multiplier architecture for Xilinx FPGAs," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, June 2015, pp. 18–25.

[8] M. Kumm, O. Gustafsson, F. de Dinechin, J. Kappauf, and P. Zipf, "Karatsuba with rectangular multipliers for FPGAs," *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pp. 13–20, 2018.

[9] M. Langhammer and B. Pasca, "Folded integer multiplication for FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 160–170. [Online]. Available: https://doi.org/10.1145/3431920.3439299

[10] M. Erle, E. Schwarz, and M. Schulte, "Decimal multiplication with efficient partial product generation," in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, 2005, pp. 21–28.

[11] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high-performance parallel decimal multipliers," in *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, 2007, pp. 195–204.

[12] S. Boldo, D. Gallois-Wong, and T. Hilaire, "A correctly-rounded fixed-point-arithmetic dot-product algorithm," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, 2020, pp. 9–16.

[13] C. Walter and D. Samyde, "Data dependent power use in multipliers," in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, 2005, pp. 4–12.

[14] K. C. Bickerstaff, E. E. Swartzlander, and M. J. Schulte, "Analysis of column compression multipliers," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ser. ARITH '01. USA: IEEE Computer Society, 2001, p. 33.

[15] N. Burgess, "The flagged prefix adder and its applications in integer arithmetic," *J. VLSI Signal Process. Syst.*, vol. 31, no. 3, pp. 263–271, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1015421507166

[16] A. Beaumont-Smith and C.-C. Lim, "Parallel prefix adder design," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 218–225.

[17] S. Knowles, "A family of adders," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 277–281.

[18] Synopsys, *DW02_mult*, 2023, https://www.synopsys.com/dw/ipdir.php?c=DW02_mult.

[19] ——, *DW02_tree*, 2023, https://www.synopsys.com/dw/ipdir.php?c=DW02_tree.