A Simple High-Speed Multiplier Design

Jung-Yup Kang, *Member*, *IEEE*, and Jean-Luc Gaudiot, *Fellow*, *IEEE*

Abstract—The performance of multiplication is crucial for multimedia applications such as 3D graphics and signal processing systems, which depend on the execution of large numbers of multiplications. Previously reported algorithms mainly focused on rapidly reducing the partial products rows down to final sums and carries used for the final accumulation. These techniques mostly rely on circuit optimization and minimization of the critical paths. In this paper, an algorithm to achieve fast multiplication in two's complement representation is presented. Rather than focusing on reducing the partial products rows down to final sums and carries, our approach strives to generate fewer partial products rows. In turn, this influences the speed of the multiplication, even before applying partial products reduction techniques. Fewer partial products rows are produced, thereby lowering the overall operation time. In addition to the speed improvement, our algorithm results in a true diamond-shape for the partial product tree, which is more efficient in terms of implementation. The synthesis results of our multiplication algorithm using the Artisan TSMC $0.13um$ 1.2-Volt standard-cell library show 13 percent improvement in speed and 14 percent improvement in power savings for 8-bit $\times$ 8-bit multiplications (10 percent and 3 percent, respectively, for 16-bit $\times$ 16-bit multiplications) when compared to conventional multiplication algorithms.

Index Terms—Multiplier, Booth, modified Booth, partial products.

1 Introduction

The performance of 3D graphics and signal processing systems strongly depends on the performance of multiplications because these applications need to support operations that are highly multiplication-intensive. Therefore, there has been much work on advanced multiplication algorithms and designs [2], [4], [5], [7], [8], [13], [14], [15], [16], [17], [19], [21], [24], [25], [26].

There are three major steps to any multiplication. In the first step, the partial products are generated. In the second step, the partial products are reduced to one row of final sums and one row of carries. In the third step, the final sums and carries are added to generate the result. Most of the above-mentioned approaches employ the Modified Booth Encoding (MBE) approach [5], [7], [8], [14], [15], [26] for the first step because of its ability to cut the number of partial products rows in half. They then select some variation of any one of the partial products reduction schemes, such as Wallace trees [7], [24] or compressor trees [14], [15], [16], [17], [19], in the second step to rapidly reduce the number of partial product rows to the final two (sums and carries). In the third step, they use some kind of advanced adder approach such as carry-lookahead or carry-select adders [6], [12], [18] to add the final two rows, resulting in the final product.

The main focus of recent multiplier papers [5], [8], [13], [17], [21], [26] has been on rapidly reducing the partial product rows by using some kind of circuit optimization and identifying the critical paths and signal races. In other words, the goals have been to optimize the second step of the multiplication described above.

- *J.-Y. Kang is with the Library and IP Group, Core Technology, Mindspeed Technology, Inc., 4311 Jamboree Rd., Newport Beach, CA 92660. E-mail: uscjungyup@yahoo.com.*
- *J.-L. Gaudiot is with the Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA 92697-2625. E-mail: gaudiot@uci.edu.*

Manuscript received 3 Apr. 2005; revised 10 Dec. 2005; accepted 27 Mar. 2006; published online 22 Aug. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0094-0405.

However, in this paper, we will concentrate on the first step, which consists of forming the partial product array, and we will strive to design a multiplication algorithm which will produce fewer partial product rows. By having fewer partial product rows, the reduction tree can be smaller in size and faster in speed. It should also be noted that 8 or 16-bit words are the most commonly used word sizes in the kernels of most multimedia applications [20] and that the implementation of our overall algorithm is particularly well suited to such word sizes.

In the next section, the conventional multiplication method is described with an emphasis on its weaknesses. In Section 3, a step-by-step procedure to prevent the adverse effects of some conventional multiplication algorithms is presented. In Section 4, the effectiveness and the implementation evaluation and analysis of our method are described and, finally, a summary will be presented.

2 The Conventional Multiplication Algorithms and the Overhead

There is no doubt that MBE is efficient when it comes to reducing the partial products. This is because, by applying MBE, the number of partial product rows to be accumulated is reduced from n to $\frac{n}{2}$ and this is why MBE is used in so many multipliers [5], [7], [8], [14], [15], [26]. However, it is important to note that there are two unavoidable consequences of using MBE: sign extension prevention and negative encoding. The combination of these two unavoidable consequences results in the formation of one additional partial product row and, of course, this additional partial product row requires not only more hardware, but also, and more importantly, time (to add this one more row of partial products).

Let us look at the benefit and the overhead of MBE by an example. For an 8-bit $\times$ 8-bit multiplication, a multiplier without MBE will generate eight partial product rows (because there is one partial product row for each bit of the multiplier). However, with MBE, only $\frac{n}{2}(=4)$ partial products rows are generated, as shown in the example of Fig. 1.

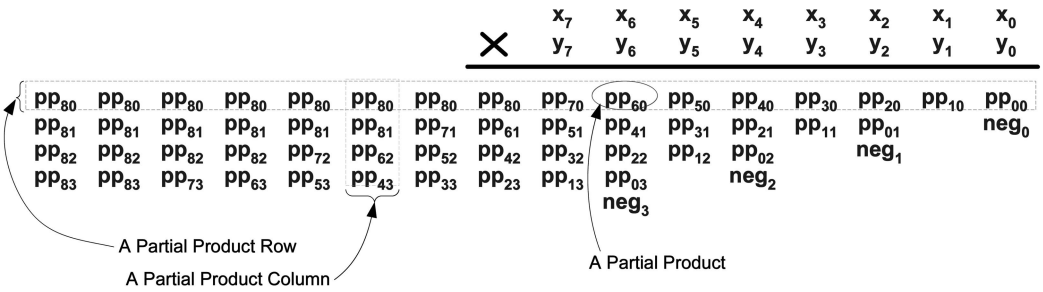

Fig. 1. The array of partial products for signed multiplication with MBE.

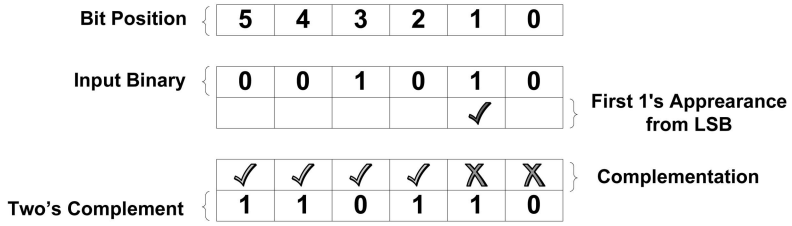

Fig. 2. Two's complement conversion example.

However, there are actually $\frac{n}{2}+1$ partial product rows rather than $\frac{n}{2}$ because of the last neg signal (neg_3 in Fig. 1). The neg signals (neg_0, neg_1, neg_2, and neg_3) are needed because MBE may generate a negative encoding ((-1) times the multiplicand or (-2) times the multiplicand).

Having one more partial product row adds at least one more EXOR-delay to the time to reduce the partial products. This fact that one additional partial product row brings delay is even more critical for multiplications of smaller words ($8 \sim 16$) than with longer operands because of the relatively higher delay effect that this additional row brings. There is also an extra hardware cost since one more carry saving adder stage hardware is necessary.

3 Preventing the Additional Partial Product Row

Therefore, our goal is to remove the last neg signal. This would prevent the extra partial product row and, thus, save the time of one additional carry save adding stage and the hardware required for the additional carry save adding. We can also generate a more regularly shaped partial product array, making it a more efficient configuration for VLSI implementation.

3.1 Removing the Last neg Signal

Indeed, if MBE encoding generated signals for only $+2\times$, $+1\times$, or $0\times$ the multiplicand, the neg signals would not be necessary (thus, there would not be the additional overhead of adding the last neg) and also the partial product rows would be perfectly parallelogram-shaped after applying Agrawal and Rao's sign extension prevention procedure [3].

We noticed that if we could somehow produce the two's complement of the multiplicand while the other partial products were produced, there would be no need for the last neg because this neg signal would have already been applied when generating the two's complement of the multiplicand. Therefore, we "only" need to find a faster method to calculate the two's complement of a binary number. Thus, an efficient logarithmic way of finding the two's complement of a binary number is the topic of the next section.

3.2 A Fast Method to Find Two's Complements

The conventional method (complement a binary number and add 1 to the complemented number) will not work for us because the propagation delay of the carry linearly increases with the word size and it would be much greater than the delay to generate the partial products. Therefore, we need a faster method.

Our method is an extension of the well-known algorithm where all the bits after the rightmost "1" in the word are complemented but all the other bits are unchanged. The two's complement of a binary number 001010_2 is 110110_2 (Fig. 2). For this number, the rightmost "1" happens in bit position 1 (the check mark position in Fig. 2). Therefore, values in bit positions 2 to 5 can simply be complemented, while values in bit positions 0 and 1 are kept unchanged.

Therefore, two's complementation now comes down to finding the conversion signals that are used for selectively complementing some of the input bits. If the conversion signal at any position is "0" (the crosses in Fig. 2), then the value is kept unchanged and, if the conversion signal is "1" (the checks in Fig. 2), then the value is complemented. The conversion signals after the rightmost "1" are always 1. They are 0 otherwise. Once a lower order bit has been found to be a "1," the conversion signals for the higher order bits to the left of that bit position should all be "1."

However, this searching for the rightmost "1" could be as time consuming as rippling a carry through to the MSB since the previous bits information must be transferred to the MSB. Therefore, we must find a method to expedite this detection of the rightmost "1."

As we will see, this search for the rightmost "1" can be achieved in logarithmic time using a binary search tree-like structure. We first find the conversion signals for a 2-bit

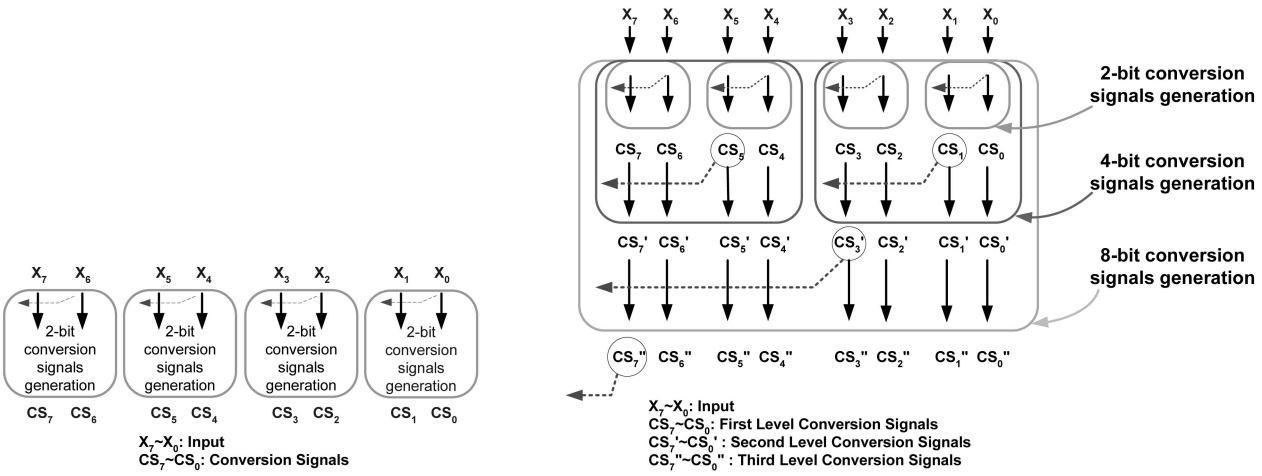

Fig. 3. Finding two's complement conversion signals. (a) Finding two's complement conversion signals for two bits. (b) Finding two's complement conversion signals for eight bits.

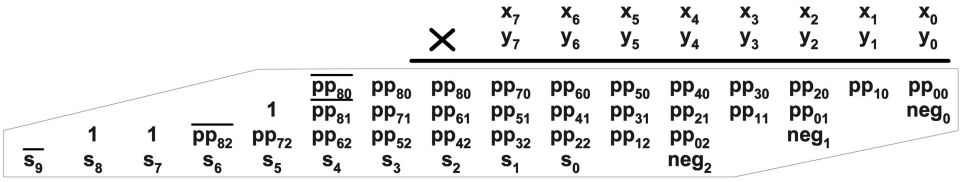

Fig. 4. Proposed partial products after removing the last neg.

group by grouping two consecutive bits (the grouping always starts from the LSB) from the input and finding the conversion signals in each group, as shown in Fig. 3a. Then, we find the conversion signals for a 4-bit group (formed by two consecutive 2-bit groups). Then, we find the conversion signals for a 8-bit group (formed by two consecutive 4-bit groups). This divide-and-conquer approach is pursued until the whole input has been covered.

When grouping two 2^n-bits groups, the leftmost conversion signals from the right group contain the accumulative information of its group about whether a "1" ever appeared in any bit position of its group so that a conversion signal should force all the conversion signals from the left group all the way to the "1" if it is itself a "1." For instance, as shown in Fig. 3b, if CS_1 (the leftmost conversion signal from the right group) = "1," the conversion signals from the left group (CS_2 and CS_3) should be forced to a "1," regardless of their previous values. If $CS_1 = "0,"$ nothing happens to the conversion signals from the left group. This variable control is shown with a dashed arrow. Likewise, CS_5 may affect conversion signals CS_6 and CS_7. The same goes for CS'_3, which may affect the conversion signals (CS'_7, CS'_6, CS'_5, and CS'_4).

3.3 Putting It All Together

By applying the method we just described for two's complementation, the last partial product row (in Fig. 1) is correctly replaced without the last neg (as in Fig. 4). Now, the multiplication can have a smaller critical path. This avoids having to include one extra carry saving adding stage. It also reduces the time to find the product and saves the hardware corresponding to the carry saving adding stage.

The multiplier architecture to generate the partial products is shown in Fig. 5. The only difference between our architecture and the conventional multiplier architectures is that, for the last partial product row, our architecture has no partial product generation but partial product selection with a two's complement unit. The 3-5 coder selects the correct value from five possible inputs ($2 \times$ X, $1 \times$ X, 0, $-1 \times$ X, $-2 \times$ X) which are either coming from the two's complement logic or the multiplicand itself and input into the row of 5-1 selectors. Unlike the other rows which use PPRG (Partial Product Row Generator), the two's complement logic does not have to wait for MBE to finish: The two's complementation is performed in parallel with MBE and the 3-5 decoder.

4 Critical Path and Performance Analysis

The performance of our multiplier architecture clearly depends on the speed of the two's complementation step. If we can generate the last partial product row of our multiplier architecture within the exact time that the other partial product rows are generated, the performance will be improved as we have predicted because of the removal of the additional partial product row.

Therefore, in this section, first, we look at the comparison of the delays of our method of generating the last partial product row to the the delays of the conventional partial product generation method (by using normalized gate delays). Then, we investigate the overall performance (in terms of speed, area, and power) of using our multiplier architecture as compared to the conventional methods. We do this by investigating the implementation-level performance

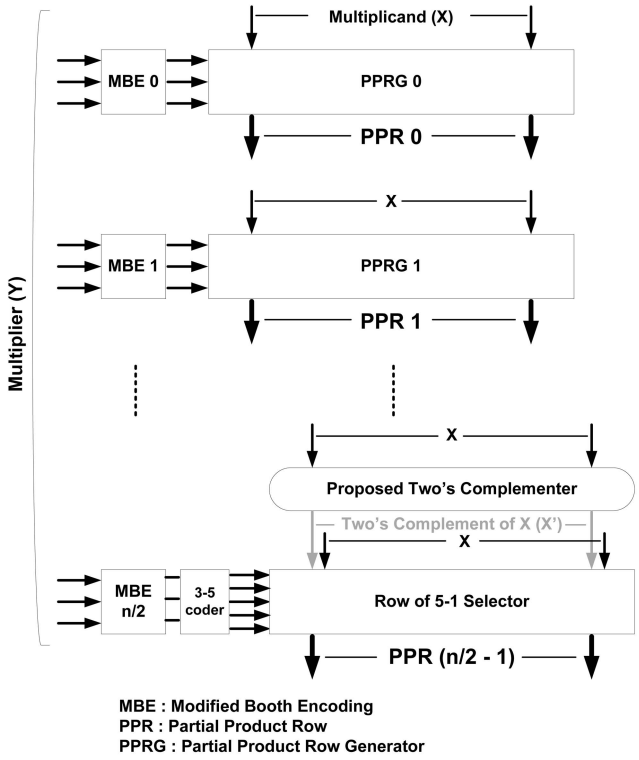

Fig. 5. Proposed multiplier architecture.

of the proposed multiplication algorithm. We designed our algorithms using Verilog HDL (Hardware Description Language) and synthesized it using Synopsys synthesis tools [22]. Note that we used Artisan TSMC $0.13um$ 1.2-Volt standard-cell library [1] with "slow corner" operating conditions for our synthesis. We also show the implementation-level performance of our two's complementation algorithm.

4.1 Critical Path Delay Comparison

The delays for the generation of the other partial products and for our last partial product row selection are shown in Fig. 6. We have used the concept of normalized gate delays to compare the delays. Although normalized gate delays vary with process technology and designs, this approach has often been used [17], [21], [23], [26] to estimate the performance of logic units. There have also been several reports on estimated normalized gate delays [9], [12], [26]. We observe that Gajski [9] has comprehensively examined these delays and the hardware cost for many different gates and combinations. Therefore, we have selected his analysis as a guideline to estimate performance.

As can be seen in Fig. 6, for 16 or 32-bit multiplications, the delay of two's complementation renders our selection method slower (**A** is smaller than any of **E**, **F**, or **G**). However, as we have mentioned before, there is the delay of one additional EXOR-gate (or up to one additional carry save adding (**C**)) due to the additional partial product (**B**); therefore, **E**, **F**, and **G** still are faster than the conventional method. If we take into account the fact that a signal for a 4-2 compressor [17] may arrive one EXOR-gate delay late, the delays of **E**, **F**, and **G** can be masked as well (this portion is highlighted with a dotted box in Fig. 6).

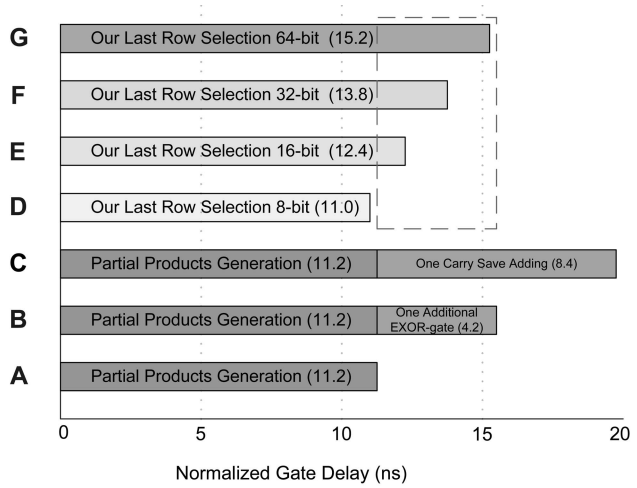

Fig. 6. Delay comparison of our last partial product row selection with other partial products generations.

Besides the speed advantage, our architecture has one other advantage: It is easily expandable for multipliers that are in sizes of power of 2. This is the case because, by using 4-2 compressors, our multipliers can naturally receive partial product rows in numbers of power of 2 (4, 8, 16, 32, 64, and so on). However, in the conventional schemes [11], [15], [17] (which have one extra partial product row), in order to obtain better performance, complex combinations of different reductions cells (using 4-2 compressors, half adders, and full adders) and different wiring schemes for different sized data words are used. This fact shows that our multiplier architecture is regular and that it can be easily modularized and, therefore, our architecture is more efficient for multiplier module generations for multipliers based on the power of 2 (which is the natural size for computing data words).

4.2 The Performance of the Proposed Two's Complementation Implementations

We have implemented our proposed 8-bit two's complement logic in Fig. 3 (of 8, 16, 32, 64, and 128-bit two's complementations) and, in order to compare the performance (against the conventional method using an adder), we also implemented a two's complementation logic using a CLA (Carry-Lookahead Adder in [6]) (we used a high-speed CLA for the conventional method instead of a ripple carry adder in order to be as fair as possible in our evaluation).

Our synthesis results (Table 1) show that both methods result in linear growth in area and power as the input size increases. The delays for both methods show somewhat logarithmic characteristics. However, when the methods are compared, our approach brings up to $2.8\times$ (when $n = 8$) performance improvement, up to $2.47\times$ (when $n = 8$) area savings, and up to $7.27\times$ (when $n = 128$) power saving when compared to the conventional method. We notice that, as we increase the input size, the improvements from delay and area shrink. When $n = 128$, we achieve about $1.54\times$ performance improvement and about $1.52\times$ area saving. We believe this phenomenon is due to the fact that, as we increase the size of the operator, the fan-out of the last stage OR-gate is severely impacted, which results in greater delays and area penalty. The power savings (in percentage) are about the same across the input sizes.

TABLE 1
Synthesis Reports of the Proposed Two's Complementation Method

	Input (*n-bit*)	8-bit	16-bit	32-bit	64-bit	128-bit
Our Method	Delay (*ns*)	0.47	0.71	1.06	1.62	2.17
	Area (um^2)	201	480	1111	2528	5661
	Power (*mW*)	0.096	0.18	0.35	0.69	1.37
Conventional (using CLA)	Delay (*ns*)	1.33	1.64	2.40	2.80	3.35
	Area (um^2)	497	997	2122	4274	8614
	Power (*mW*)	0.51	1.04	2.35	4.74	9.96

TABLE 2
Synthesis Reports of the Proposed Multiplication Algorithm

Multiplication Size		8-bit x 8-bit	16-bit x 16-bit
Our Method	Delay (*ns*)	3.03	4.42
	Area (um^2)	4030	14859
	Power (*mW*)	6.3876	25.2736
Conventional	Delay (*ns*)	3.42	4.84
	Area (um^2)	4578	14840
	Power (*mW*)	7.2756	25.9729

4.3 The Performance of the Proposed Multiplication Algorithm Implementations

We have used the multiplier architecture in Fig. 5 as our implementations (of 8-bit $\times$ 8-bit multiplier and 16-bit $\times$ 16-bit multiplier). In order to compare the performance, we have chosen the most efficient multiplier architectures (in terms of the delay to reduce the partial products down to final sums and carries) for the conventional methods. From our analysis, we learned that the optimized Wallace tree scheme brings the shortest delay for the conventional 8-bit $\times$ 8-bit and 16-bit $\times$ 16-bit multiplications and, therefore, we have used the optimized Wallace tree scheme for the conventional multiplier architectures. Except for the fact that the final partial product row is generated differently, the other partial products are generated similarly for both our and the conventional implementations and both implementations use carry select adders for the final addition.

Our synthesis implementation results show 13 percent speed improvement and 14 percent power savings for 8-bit $\times$ 8-bit multiplications and 10 percent speed improvement and 3 percent power savings for 16-bit $\times$ 16-bit multiplications when compared to the conventional multiplication algorithms. The exact performance of the implementations is shown in Table 2. We have saved about 14 percent area for 8-bit $\times$ 8-bit multiplications, but used slightly more area (a little more than 1 percent) for 16-bit $\times$ 16-bit multiplications.

4.4 Related Work

Previous work [7], [17], [21], [26] has shown that the critical path of the multiplications was the reduction of the partial products from the $(n-2)\mathrm{nd}$ to the $n\mathrm{th}$ bit positions because these bit positions have the highest number of partial products. Ercegovac and Lang [6] have pointed out that the presence of an extra partial product row was due to the last neg signal. Huang and Ercegovac [11] endeavored to overcome the presence of this additional row by devising a 9-4 compressor whose goal it is to reduce nine partial product rows within the same time that two 4-2 compressors are reducing eight partial product rows. This has the effect of masking the latency due to the reduction of one additional partial product row. However, this approach is only specific to his design of 9-4 reduction (and his assumption about inputs being constants) and has no flexibility. On the other hand, our scheme just plugs two's complementation logic before selecting the last partial product row and it is therefore easier to implement and can be applied to any bit length multiplications.

Again, all this related work is based on some kind of efficient circuit optimization technique to reduce $\frac{n}{2}+1$ partial product rows (the second step), whereas our algorithm is to concentrate on the first step, which consists of forming the partial product array. By having fewer partial product rows, the reduction tree can be smaller in size and faster in speed. Furthermore, since our algorithm is targeted at forming the tree rather than reducing it, recent circuit optimization techniques [5], [8], [13], [17], [21], [26] for its reduction can be easily incorporated with our method to further improve the performance.

Related to our two's complementation method, Hwang [12] and Hashemian and Chen [10] have shown similar approaches (finding the conversion signals). However, our method is logarithmic, whereas Hwang's method is linear and Hashemian and Chen focused on circuit optimization to improve the performance. Our approach is more general and shows better adaptability to any word size.

5 Conclusions

In this paper, we have presented a simple, high-speed, and well-structured multiplication algorithm. Our multiplication algorithm focuses on the first step of a multiplication algorithm, which is the partial product generation step to reduce from $\frac{n}{2}+1$ to $\frac{n}{2}$ the number of partial product rows generated. By doing so, the structure of the partial product array becomes more regular and easier to implement. Even more importantly, the product is found faster because of the smaller number of partial product rows to add. As we have shown, we can achieve this using less hardware.

We have shown a detailed and step-by-step approach to prevent the occurrence of the additional row and also showed a generalized way of achieving two's complementation in logarithmic time. We also have shown how our algorithm can exploit the characteristics of the current designs of the 4-2 compressors in order to be efficiently implemented. We have also shown that the proposed multiplication method is particularly efficient when executing the multiplications of the kernels of the most common multimedia applications, which are based on 8 to 16-bit operands.

The synthesis results of the proposed two's complementation and multiplication methods show that our two's complementation method also brings $2.8\times$ (8-bit) and $2.31\times$ (16-bit) performance improvements and $5.31\times$ (8-bit) and $5.78\times$ (16-bit) power savings when compared to the conventional method. Our multiplication algorithm shows 13 percent speed improvement and 14 percent power savings for 8-bit $\times$ 8-bit multiplications and 10 percent speed improvement and 3 percent power savings for 16-bit $\times$ 16-bit multiplications when compared to the conventional multiplication algorithms.

ACKNOWLEDGMENTS

This paper is based upon work supported in part by US National Science Foundation (NSF) Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Artisan Components, *TSMC 0.13μm Process CL013LV 1.2-Volt SAGE-X Standard Cell Library Databook.* Artisan Components, Oct. 2001.
[2] A.D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical and Applied Math.,* vol. 4, pp. 236-240, 1951.
[3] D.P. Agrawal and T.R.N. Rao, "On Multiple Operand Addition of Signed Binary Numbers," *IEEE Trans. Computers,* vol. 27, pp. 1068-1070, Nov. 1978.
[4] L. Dadda, "Some Schemes for Parallel Multiplier," *Alta Frequenza,* vol. 34, pp. 349-356, 1965.
[5] F. Elguibaly, "A Fast Parallel Multiplier-Accumulator Using the Modified Booth Algorithm," *IEEE Trans. Circuits and Systems,* vol. 47, no. 9, pp. 902-908, 2000.
[6] M.D. Ercegovac and T. Lang, *Digital Arithmetic.* Los Altos, Calif.: Morgan Kaufmann, 2003.
[7] J. Fadavi-Ardekani, "M x N Booth Encoded Multiplier Generator Using Optimized Wallace Trees," *IEEE Trans. Very Large Scale Integration,* vol. 1, no. 2, pp. 120-125, 1993.
[8] A. Farooqui and V. Oklobdzija, "General Data-Path Organization of a MAC Unit for VLSI Implementation of DSP Processors," *Proc. 1998 IEEE Int'l Symp. Circuits and Systems,* vol. 2, pp. 260-263, 1998.
[9] D. Gajski, *Principles of Digital Design.* Prentice Hall, 1997.
[10] R. Hashemian and C. P. Chen, "A New Parallel Technique for Design of Decrement/Increment and Two's Complement Circuits," *Proc. 34th Midwest Symp. Circuits and Systems,* vol. 2, pp. 887-890, 1991.
[11] Z. Huang and M. Ercegovac, "High-Performance Left-to-Right Array Multiplier Design," *Proc. 16th Symp. Computer Arithmetic,* pp. 4-11, June 2003.
[12] K. Hwang, *Computer Arithmetic Principles, Architecture, and Design.* New York: Wiley, 1979.
[13] N. Itoh, Y. Naemura, H. Makino, Y. Nakase, T. Yoshihara, and Y. Horiba, "A 600-MHz 54x54-bit Multiplier with Rectangular-Styled Wallace Tree," *IEEE J. Solid-State Circuits,* vol. 36, no. 2, pp. 249-257, 2001.
[14] J.-Y. Kang and J.-L. Gaudiot, "A Fast and Well-Structured Multiplier," *EUROMICRO Symp. Digital System Design,* pp. 508-515, Aug. 2004.
[15] J.-Y. Kang, W.-H. Lee, and T.-D. Han, "A Design of a Multiplier Module Generator Using 4-2 Compressor," *Proc. Korea Inst. of Telematics and Electronics (KITE) Fall Conf.,* vol. 16, pp. 388-392, 1993.
[16] M. Nagamatsu, S. Tanaka, J. Mori, T. Noguchi, and K. Hatanaka, "A 15ns 32x32-bit CMOS Multiplier with an Improved Parallel Structure," *Digest of Technical Papers, IEEE Custom Integrated Circuits Conf.,* 1989.
[17] V.G. Oklobdzija, D. Villeger, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers* vol. 45, no. 3, pp. 294-306, Mar. 1996.
[18] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach.* San Mateo, Calif.: Morgan Kaufmann, 1996.
[19] M.R. Santoro and M. Horowitz, "SPIM: A Pipelined 64x64-bit Iterative Multiplier," *IEEE Trans. Circuits and Systems,* vol. 24, no. 2, pp. 487-493, 1989.
[20] N. Slingerland and A.J. Smith, "Measuring the Performance of Multimedia Instruction Sets," *IEEE Trans. Computers,* vol. 51, no. 11, pp. 1317-1332, 2002.
[21] P.F. Stelling, C.U. Martel, V.G. Oklobdzija, and R. Ravi, "Optimal Circuits for Parallel Multipliers," *IEEE Trans. Computers,* vol. 47, no. 3, pp. 273-285, Mar. 1998.
[22] Synopsys. *Design Compiler User's Guide,* http://www.synopsys.com/, 2004.
[23] D. Villeger and V. Oklobdzija, "Analysis of Booth Encoding Efficiency in Parallel Multipliers Using Compressors for Reduction of Partial Products," *Proc. 27th Ann. Asilomar Conf. Signals, Systems, and Computers,* vol. 1, pp. 781-784, 1993.
[24] C.S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. Computers,* vol. 13, no. 2, pp. 14-17, 1964.
[25] A. Weinberger, "4:2 Carry-Save Adder Module," *IBM Technical Disclosure Bull.,* vol. 23, 1981.
[26] W.-C. Yeh and C.-W. Jen, "High-Speed Booth Encoded Parallel Multiplier Design," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 692-701, July 2000.

Jung-Yup Kang received the BS and MS degrees in computer science from Yonsei University, Seoul, Korea, in 1992 and 1994, respectively, and he received the MS and PhD degrees in computer engineering from the University of Southern California, Los Angeles, in 1999 and 2004, respectively. He is currently a senior staff design automation engineer in the Core Technology Group at Mindspeed Technology, Inc., in Newport Beach, California. He also worked for Synopsys as a professional service consultant (1996-1997), worked as a research staff member at Samsung Semiconductor Co. designing the floating-point unit for a 64-bit microprocessor (1994-1996), and worked as a design engineer at MOSIS/ISI/USC (1998-2000). His research interests include computer architecture, computer arithmetic, parallel processing, and VLSI designs. He is a member of the IEEE.

Jean-Luc Gaudiot received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France, in 1976 and the MS and PhD degrees in computer science from the University of California, Los Angeles, in 1977 and 1982, respectively. He is currently a professor and chair of the Department of Electrical Engineering and Computer Science at the University of California, Irvine (UCI). Prior to joining UCI in January 2002, he had been a professor of electrical engineering at the University of Southern California since 1982, where he served as director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979-1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980-1982). His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published more than 170 journal and conference papers. He served as editor-in-chief of the *IEEE Transactions on Computers* (1999-2002) and has been the editor-in-chief of the *IEEE Computer Architecture Letters* since January 2006. He is a member of the ACM, of ACM SIGARCH, and a fellow of the IEEE.