Synthesizable Verilog Code Generator for Variable-Width Tree Multipliers

Article *in* Journal of Physics Conference Series · July 2021

DOI: 10.1088/1742-6596/1962/1/012046

2 authors, including:

Nandha Kumar Thulasiraman
University of Nottingham, Malaysia Campus
112 PUBLICATIONS **831** CITATIONS

Synthesizable Verilog Code Generator for Variable-Width Tree Multipliers

Synthesizable Verilog Code Generator for Variable-Width Tree Multipliers

Chuah Ching Fun, Nandha Kumar Thulasiraman

Dept. of Electrical and Electronic Eng. University of Nottingham Malaysia
Email: {kecy6ccf or nandhakumaar.t}@nottingham.edu.my

Abstract. Tree multipliers are fast multipliers which are important for timing-critical applications. However, due to the irregular multiplier structure, the process of coding a tree multiplier is often very time-consuming. In addition, it is difficult to generalize the multiplier codes for variable-width inputs. In this paper, the authors used Python scripts to generate Verilog codes for tree multipliers automatically in a very short amount of time, by specifying only the required data width. The generated tree multiplier designs are synthesized and implemented using TSMC 180nm technology as well as Artix-7 FPGA. Through analysis, it is observed that modified Booth multiplier designed with Dadda tree reduction algorithm has up to 47% smaller area and up to 71% shorter delay compared to array multiplier. All multiplier designs are thoroughly simulated and functionally verified.

1. Introduction

Multiplier is a common yet crucial component of arithmetic and logic units in computer systems. Since multipliers often play vital roles in high-performance applications such as image processing and digital signal processing [1][2], it is generally desired for multiplier designs to occupy small hardware area, consume low power, and incur minimal amount of delay, in order to achieve acceptable level of overall system performance. However, multiplier is usually the largest combinational circuit [3] in a system and thus limits the maximum applicable system clock frequency, which in turn puts a constraint on the system throughput achievable.

Tree multiplier is a widely acknowledged fast multiplier structure [4], establishing high-speed performance with some trade-offs in terms of hardware area. Hence tree multipliers are frequently employed in various system and processor designs. Nevertheless, one main disadvantage of tree multiplier is its irregular structure. This poses considerable amount of challenge in optimizing a tree multiplier design because the time delay for every individual signal path does not follow an easily analyzable pattern. Moreover, it is also very tedious to code a tree multiplier design in a hardware description language (HDL). All the adder blocks and individual bits of the intermediate results have to be coded one by one to correctly describe the wire connections. Unlike regular multiplier structures such as array multiplier, it is very difficult to reuse the same piece of code for the same tree multiplier design with a different data width, by just changing a parameter that specifies the number of bits. Instead, most of the coding process has to be repeated if a different data width is required. Obviously, this will take up a lot of time and effort in the design and coding process. The resulting piece of code is also prone to human errors which lead to unnecessary work to search for errors when the design does not pass validation tests. The proposed method helps to rapidly develop the adder circuits of any length for the

applications such as approximate computing, image processing, Field programmable gate array testing, etc[6]-[9].

In this paper, the authors present a method to overcome the tedious coding process of tree multipliers by using automatic generator for the tree multiplier codes in HDL. The scripting language used is Python and the generated tree multiplier codes are in Verilog, an HDL of industrial standard. By just providing the data width, the complete and synthesizable Verilog codes for a tree multiplier design can be generated instantly. The whole generation process takes less than one second which is much more efficient over typing the codes for tree multipliers line by line. Furthermore, it can be said that there will not be any Verilog coding error since the codes are generated in a predefined and systematic way, rather than being keyed in manually. The authors also verified the correct functionalities of all the tree multipliers. The multiplier codes are accurate and ready for synthesis upon generated without any manual alteration required. The authors synthesized and implemented each tree multiplier design using TSMC 180nm technology as well as Xilinx Artix-7 FPGA. These multiplier designs are analyzed and compared for their performances.

This paper is structured as follows. Section II introduces some popular multiplier designs [5] including tree multipliers. Section III describes in detail the Python scripts which are used to generate Verilog codes. After that, Section IV presents the experimental results using the script-generated tree multiplier designs. Lastly Section V concludes this paper.

2. Multiplier

2.1. Array Multiplier

Array multiplier is a regular multiplier structure in which the full adder and half adder blocks can be coded easily using for loops in Verilog. In the case of array multiplier, the Verilog codes can include a general parameter that configures the data width of the multiplicands. Hence the same piece of code can be reused for varied data widths. Fig. 1 shows a general array multiplier structure to reduce the ANDed partial products.

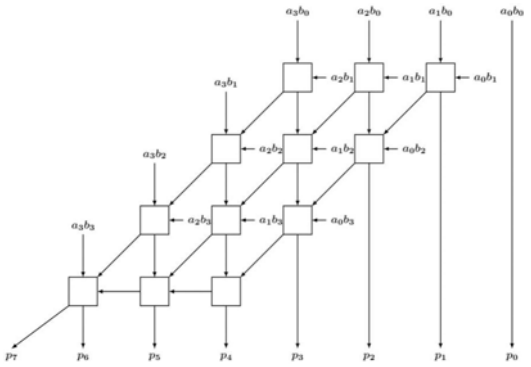


Figure 1.4-bit array multiplier, where a and b are two multiplicands while p is their product.

2.2. Tree multiplier

Tree multipliers consist of three phases, namely partial product generation, partial product reduction, and carry propagate adder [10][11]. Partial products are generated by ANDing each bit of a multiplicand with every bit of the other multiplicand. Then the partial product matrix is reduced by full adder and half adder blocks into two rows, which are finally summed up to obtain the product. Examples of tree multipliers are Wallace tree multiplier [12] and Dadda tree multiplier [13]. They differ in the algorithms used at the partial product reduction phase. In the greedier Wallace algorithm, as many adders as possible are used

in each column of the partial product matrix. On the other hand, minimum number of adders are used in Dadda algorithm.

Conventionally, Wallace tree multiplier is designed by first dividing the partial product matrix into groups of three rows. Then within each group, any column with three bits is summed up using a full adder, whereas any column with two bits is summed up using a half adder. The grouping and adding process is repeated for as many stages as required until there are only two rows remaining in the partial product matrix. This is illustrated in Fig.2(a).

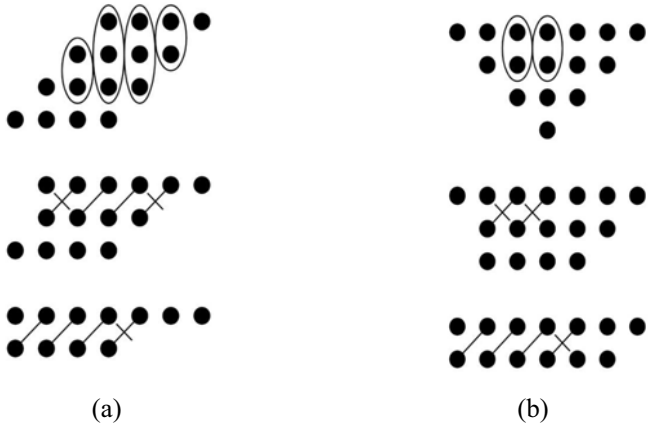


Figure 2. Dot representation for 4-bit (a) Wallace tree multiplier reduction process, in which 5 full adders and 3 half adders are used, (b) Dadda tree multiplier reduction process, in which 3 full adders and 3 half adders are used.

As for Dadda tree multiplier, at each stage the least number of adder blocks are used to reduce the height of the partial product matrix into a certain maximum height. The maximum height at each stage can be calculated backwards from the final stage, for which the height is 2. Each stage should have a maximum height of $\lfloor 1.5h \rfloor$ where h is the maximum height of the successive stage. For example, the maximum height of second last stage is 3. Therefore in Fig. 2(b), the initial partial product matrix is reduced into 3 rows by using two half adders. Any column with height equal to or less than the maximum height is not reduced. It can be seen that for different widths of multiplicands, the resulting positions of adder blocks are different according to the applied algorithm. Hence tree multipliers have irregular structures which cannot be coded in general terms.

2.3. Modified Booth Multiplier

Modified Booth multiplier differs from the other multiplier structures in its partial product generation [14]. In a Modified Booth multiplier, one of the multiplicands is radix-4 Booth encoded. As a result, the number of rows of partial products generated is halved. By simplifying the extended sign bits, the number of partial product bits becomes approximately half that of an ordinary multiplier. Fewer adder blocks are required to reduce the partial product tree. Hence the signal paths for partial product reduction phase are shorter, resulting in faster performance. On top of that, modified Booth multiplier can readily carry out signed multiplication without further logics. The partial product reduction phase of Modified Booth multiplier is normally done by tree reduction algorithms. Wallace algorithm and Dadda algorithm are applied as before, but on a different partial product matrix. Thus, it would also be great if HDL codes for Modified Booth multipliers can be generated automatically by running scripts.

3. Verilog Code Generator

In this paper, the Verilog code generation scripts of Wallace and Dadda tree multipliers are written using Python scripting language, utilizing various string processing functions to construct the Verilog statements automatically. Besides, the Python list of lists data structure is used to store elements of the partial product matrix at each stage of the reduction process.

Since full adder and half adder blocks are frequently needed in the partial product reduction phase, they are first coded as separate Verilog modules to be instantiated later in the tree multiplier module. For both Wallace and Dadda tree multipliers, the code generation process is similar. A partial product matrix is formed by populating a Python list of lists with Verilog identifiers corresponding to the partial product bits. The complete partial product matrix is then reduced according to the chosen algorithm. During the reduction process, elements in the Python list of lists are modified, and lines of Verilog statements which instantiate adder blocks are appended to a string variable holding the Verilog codes. For example, when a half adder is to be instantiated, two elements are deleted from certain column of the partial product matrix while a new sum bit and a new carry bit are inserted at suitable column positions. At the same time, one line of Verilog statement defining a half adder is appended to the accumulated Verilog codes. This is continued until two rows are left in the partial product matrix. Finally, the Verilog code for a carry propagate adder is included to sum up the remaining elements in the last two rows. In this work, the carry propagate adder is coded by concatenating the bits in each of the last two rows and joining them with a '+' operator. This is to utilize the tool-optimized adder designs of different implementation technologies. Nevertheless, this adder can be replaced by some specific fast adder designs such as carry lookahead adder [15].

The Verilog codes for partial product generation and miscellaneous Verilog declarations are put before the codes for partial product reduction and carry propagate adder. With that, a synthesizable tree multiplier design is completed. For each of Wallace and Dadda algorithms, two versions of the code generation scripts are written, one for the ordinary tree multiplier, another for Modified Booth multiplier. Each version has a different set of Verilog codes for partial product generation and elements in the partial product matrix. A user of the code generation scripts is only required to provide the data width of the multiplicands using command line interface to generate the complete Verilog codes for tree multipliers, which may go above a few hundred lines, in a very short amount of time. Although the scripts are written for Verilog codes in this work, similar approach can be applied for other HDLs as well. The following parts describes the logic flow in partial product reduction phase for both Wallace and Dadda tree multipliers.

3.1. Wallace Tree Multiplier

The partial product matrix for Wallace tree multiplier is stored in row-major in a Python list of lists because at every stage, groups of three rows have to be formed. As shown in Fig. 3, groups of three rows are formed as long as the number of rows of the partial product matrix is larger than two.

```
bit_matrix = populate_bit_matrix(n)
code = ""

while len(bit_matrix) > 2:
    for g in groups_of_three_rows(bit_matrix):
        for column in g:
            if number_of_bits(column) == 3:
                code += instantiate_FA(column)
            elif number_of_bits(column) == 2:
                code += instantiate_HA(column)
            remove_and_insert_bits(column)
    remove_empty_rows(bit_matrix)
code += "assign P = {} + {};".format(
    ",".join(bit_matrix[0]), ",".join(bit_matrix[1])
)

return code
```

Figure 3. Pseudocode to generate Verilog code for *n*-bit Wallace tree multiplier.

After reducing those columns with two bits or three bits using adders, the remaining bits in each group are routed to the next stage together with the newly obtained carry bits and sum bits. Note that the resulting carry bits and sum bits for the current stage are not taken into account when counting the height of each column. Any resulting empty rows are deleted from the partial product matrix. This is repeated until end of the final stage where there are only two rows left.

3.2. Dadda Tree Multiplier

On the other hand, the partial product matrix for Dadda tree multiplier is populated in column-major as the height of each column has to be kept updated. After determining the total number of stages required for a certain data width and obtaining the list of maximum heights for each stage, a for loop is defined to iterate the reduction process as many times as the required number of stages, as shown in Fig. 4. If a column has a greater number of bits than the maximum height for a particular stage, an adder block is inserted, otherwise no action is taken. A column with only one bit more than the maximum height will be reduced using a half adder instead of a full adder. Counting the height of a column for Dadda algorithm is slightly different from Wallace algorithm because the carry bits from the neighboring column and the sum bits in that column itself have to be considered, so that the total height including the carry bits and sum bits will not exceed the maximum height. After the final stage, the remaining bits in the last two rows are summed up as usual.

```
bit_matrix = populate_bit_matrix(n)
code = ""

for stage in range(number_of_stages_needed(n)):
    for column in bit_matrix:
        while height(column) > max_height(stage):
            if height(column) == max_height(stage) + 1:
                code += instantiate_HA(column)
            else:
                code += instantiate_FA(column)
            remove_and_insert_bits(column)
code += "assign P = {} + {};".format(
    ",".join(row_0(bit_matrix)), ",".join(row_1(bit_matrix))
)

return code
```

Figure 4. Pseudocode to generate Verilog code for *n*-bit Dadda tree multiplier.

4. Results and Discussion

Using the Verilog code generation scripts, both Wallace and Dadda tree multiplier designs are generated for 4-bit, 8-bit, 16-bit, and 32-bit data width. For each type of tree multiplier, two different partial product generation methods are used, that are ordinary partial product generation and modified Booth multiplier, resulting in two distinct designs. Simulations are carried out for all these multipliers and they are verified for their functionalities.

Then, each of these designs are synthesized and implemented using TSMC 180nm technology and Xilinx Artix-7 FPGA to investigate their performances in terms of area and delay. For comparison purpose, array multipliers are also designed, simulated, synthesized and implemented for various data widths.

4.1. Area Analysis

Table I shows the area of each multiplier design implemented using TSMC 180nm technology. On the other hand, the area of multiplier designs implemented using Xilinx Artix-7 FPGA is recorded in Table II. From both Table I and Table II, it can be seen that the areas of tree multipliers tend to be smaller than that of array multipliers, especially when applied together with radix-4 Booth encoding. Generally, the modified Booth multiplier designs

have smaller area than the tree multipliers with same data widths. Wallace algorithm results in larger multipliers compared to Dadda algorithm as there are more adder blocks.

Table 1. Area of Multiplier Designs Implemented Using TSMC 180nm Technology.

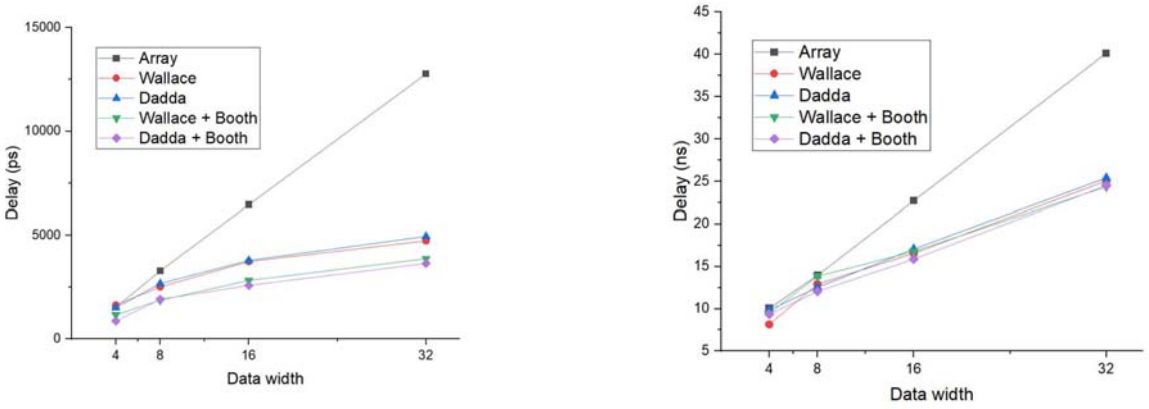
Impl.	Multiplier	Data width			
		4-bit	*8-bit*	*16-bit*	*32-bit*
		Area (μm^2)			
TSMC 180nm	Array	3844	17315	71224	298395
	Wallace	4474	17920	73683	388712
	Dadda	4114	17718	72248	297882
	Wallace + Booth	4294	16898	63914	255841
	Dadda + Booth	4294	15757	59617	247573

Table 2. Area of Multiplier Designs Implemented Using Xilinx Artix-7 FPGA.

Impl.	Multiplier	Data width			
		4-bit	*8-bit*	*16-bit*	*32-bit*
		Area (Number of LUTs)			
FPGA	Array	15	72	426	1977
	Wallace	18	77	380	1484
	Dadda	15	73	328	1315
	Wallace + Booth	16	71	300	1206
	Dadda + Booth	16	66	260	1050

4.2. Delay Analysis

Tree multipliers are faster than array multipliers. This is depicted through Fig. 5(a) and Fig. 5(b) in which delays for various multipliers implemented using TSMC 180nm technology and Xilinx Artix-7 FPGA are recorded respectively. For array multipliers, the delay increases in a linear trend with respect to data width. However, the delay for tree multiplier increases logarithmically with respect to data width, which is at a slower rate compared to array multiplier.



(a) (b)

Figure 5. Delay for various multiplier designs implemented using (a) TSMC 180nm technology (b)Xilinx Artix-7 FPGA.

Besides, modified Booth multipliers generally have shorter delay than the corresponding ordinary tree multipliers for the same data widths. On the other hand, Dadda tree multipliers tend to be a little faster than Wallace tree multipliers. Overall, the delay for modified Booth multipliers applied together with Dadda tree reduction is the shortest, showing that they are the fastest multipliers among the analyzed designs. Combined with area analysis, it can be said that out of the multiplier designs investigated in this paper, modified Booth multiplier with Dadda tree reduction is the most favorable because it has the smallest area and the shortest delay.

5. Conclusion

In this paper, various tree multiplier designs including Wallace tree multiplier, Dadda tree multiplier, and modified Booth multiplier are coded in Verilog through automated code generation scripts to eliminate the tedious process of coding these irregular multiplier structures. All the tree multipliers generated are simulated and verified to be functionable. Thus, this method of coding multiplier designs is proven to be applicable and advantageous as it is time-saving and error-free. The tree multiplier designs are also synthesized and implemented using TSMC 180nm technology and Xilinx Artix-7 FPGA to make comparison with the easily-coded regular array multipliers. It is found that modified Booth multiplier with Dadda tree reduction is the smallest as well as the fastest multiplier among the analyzed designs.

References

[1] Aravind Babu S, Babu Ramki S, and Sivasankaran K, “Design and implementation of high speed and high accuracy fixed-width modified booth multiplier for DSP application,” 2014 International Conference on Advances in Electrical Engineering (ICAEE), Vellore, 2014, pp. 1-5, doi: 10.1109/ICAEE.2014.6838565.

[2] CF Chuah, TN Kumar, “Fast and exact multiple-input unary-to-binary multiplier with variable precision for stochastic computing” IET Electronics Letters, 2020.

[3] B. Dinesh, V. Venkateshwaran, P. Kavinmalar, and M. Kathirvelu, “Comparison of regular and tree based multiplier architectures with modified booth encoding for 4 bits on layout level using 45nm technology,” 2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE), Coimbatore, 2014, pp. 1-6, doi: 10.1109/ICGCCEE.2014.6922297.

[4] T. Arunachalam and S. Kirubaveni, “Analysis of high speed multipliers,” 2013 International Conference on Communication and Signal Processing, Melmaruvathur, 2013, pp. 211-214, doi: 10.1109/iccsp.2013.6577045.

[5] K. L. S. Swee and L. H. Hiung, “Performance comparison review of 32-bit multiplier designs,” 2012 4th International Conference on Intelligent and Advanced Systems (ICIAS2012), Kuala Lumpur, 2012, pp. 836-841, doi: 10.1109/ICIAS.2012.6306130.

[6] HAF Almurib, T.Nandha Kumar, F. Lombardi, “A single-configuration method for application-dependent testing of SRAM-based FPGA interconnects” Asian Test Symposium, 201, doi: 10.1109/ATS.2011.12.

[7] HAF Almurib, TN Kumar, F Lombardi, ”Scalable application-dependent diagnosis of interconnects of SRAM-based FPGAs” IEEE Transactions on Computers, 2013, doi: 10.1109/TC.2013.34.

[8] H. Junqi, T. N. Kumar, and H. Abbas, "A promising power-saving technique: Approximate computing," in 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE), Penang, Malaysia, 2018, pp. 285-290.

[9] H. Junqi, T. N. Kumar, and H. Abbas, "Simulation-Based Evaluation of Approximate Adders for Image Processing Using Voltage Overscaling Method," in 2020 IEEE 5th International Conference on Signal and Image Processing (ICSIP), Nanjing, China, 2020, pp. 1-1.

[10] R. S. Waters and E. E. Swartzlander, "A reduced complexity Wallace multiplier reduction," in IEEE Transactions on Computers, vol. 59, no. 8, pp. 1134-1137, Aug. 2010, doi: 10.1109/TC.2010.103.
[11] J Huang, TN Kumar, H Abbas, "General Expressions for Performance Evaluation of Binary Adders" 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2020.
[12] C. S. Wallace, "A suggestion for a fast multiplier," in IEEE Transactions on Electronic Computers, vol. EC-13, no. 1, pp. 14-17, Feb. 1964, doi: 10.1109/PGEC.1964.263830.
[13] L. Dadda, "Some schemes for parallel multipliers," Alta Frequenza, vol. 34, pp. 349-356, 1965.
[14] R. Hussin, A. Y. M. Shakaff, N. Idris, Z. Sauli, R. C. Ismail, and A. Kamarudin, "An efficient Modified Booth multiplier architecture," 2008 International Conference on Electronic Design, Penang, 2008, pp. 1-4, doi: 10.1109/ICED.2008.4786767.
[15] Yu-Ting Pai and Yu-Kumg Chen, "The fastest carry lookahead adder," Proceedings. DELTA 2004. Second IEEE International Workshop on Electronic Design, Test and Applications, Perth, WA, Australia, 2004, pp. 434-436, doi: 10.1109/DELTA.2004.