Enhancement Of A 32-By-32-Bit Signed Digital Multiplier: A Multi-Dimensional Optimization Approach

Chenjia Cui [1,*], Yunpeng Li [2] and Taoyujie Shang [3]

[1] International College, Zhengzhou University, Zhengzhou, 450003, China

[2] School of Civil Engineering and Architecture, Guangxi University, Nanning, 530004, China

[3] Electronic information engineering college, Changchun University, Changchun, 130022, China

* Corresponding Author Email: cuichenjia@stu.zzu.edu.cn

Abstract. In the pursuit of enhanced performance in a 32-by-32-bit signed digital multiplier, optimizations were enacted on three critical fronts. Initial efforts focused on the integration of an advanced Booth encoding technique for the generation of three-bit control signals, meticulously crafted in the Verilog programming language. This approach, characterized by its restricted symbol expansion methodology, strategically curtails hardware resource expenditures, and diminishes power utilization during computational tasks. Subsequently, a sophisticated hybrid Wallace tree architecture was employed, offering a notable decrement in the delay of two XOR gates in contrast to the exclusive application of 3-2 and 4-2 compressors. This refinement not only augments the aggregation efficacy of partial products but also substantially accelerates operational velocity. Furthermore, the incorporation of a 64-bit carry-lookahead adder was instrumental in mitigating the latency induced by lower-order carry anticipation. To ascertain the multiplier's computational precision, a Verilog-constructed testbench file facilitated the assembly of 100,000 test vectors, with simulations executed in ModelSim yielding a verifiable accuracy rate of 100%.

Keywords: Digital Multiplier, Booth Encoding Technique, Hybrid Wallace Tree, Carry-Lookahead Adder.

1. Introduction

Multiplication operation is one of the most common and significant operations in computer systems. Especially in the era of artificial intelligence where ChatGPT is a big hit, deep learning models are an important part of artificial intelligence, which process and extract features from input data using multilayer neural networks by simulating how neurons in the human brain work to generate the corresponding output. The success of deep learning is supported by modern computer hardware and large-scale datasets [1]. In many deep learning models, multipliers are used to implement matrix multiplication. This operation multiplies the input data with a pre-trained weight matrix to obtain a new, higher-level representation.

However, the traditional multiplier design needs to multiply and sum bit by bit, so the calculation speed is slow and requires a lot of hardware resources and high energy consumption, which is unable to meet the development of the artificial intelligence era. Therefore, the optimized multiplier can be studied to significantly increase the computational speed of the multiplier, reduce the energy consumption, and reduce the hardware resource consumption, so as to achieve effective performance improvement [2].

The working process of a multiplier can be divided into partial product generation and partial product compression. Among many optimization strategies, Booth encoding, and Wallace tree have attracted much attention. Booth encoding can reduce the number of partial products, while Wallace tree can effectively compress the partial products and thus improve the performance of the multiplier.

In order to design a multiplier with optimized speed, this paper studied the structure of the multiplier improved the traditional Booth algorithm, designed a partial product array structure with arrangement rules by processing symbol expansion bits, improved the traditional 4-2 compressor, and proposed a new Wallace tree structure, and finally used a 64-bit carry-lookahead adder to complete

the calculation. Finally, Verilog code was written for verification on modelsim platform. The layout area is optimized, and the critical path delay is shortened.

2. Multiplier structure

This multiplier is designed for signed 32-bit multiplication, consisting of three main modules: partial product generation, partial product compression, and carry-lookahead adder. Its structure is shown in figure 1.

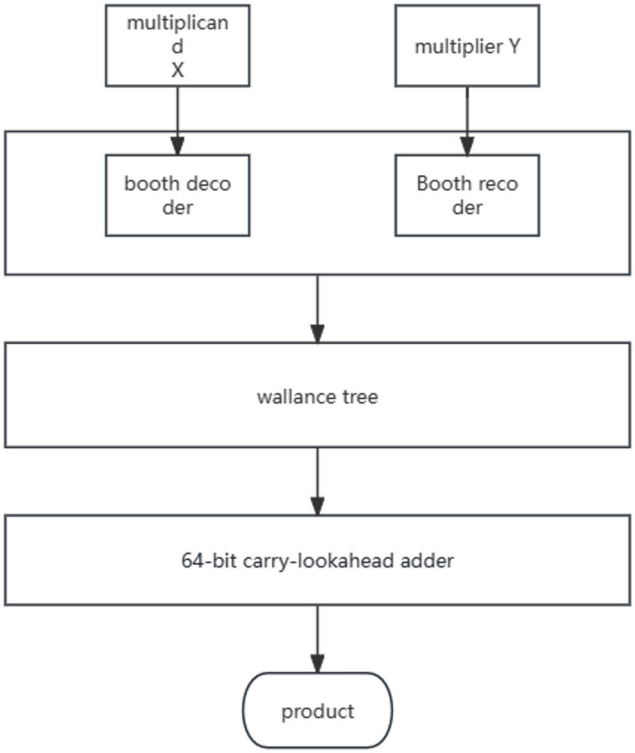

Fig. 1 Multiplier structure (Photo/Picture credit: Original)

2.1. Radix-4 Booth algorithm

Through the representation conversion between the complement of the multiplier and the true value, it can perform the following conversion.

$$N = -a_n \square 2^n + \sum_{i=0}^{n-1} a_i \square 2^i \tag{1}$$

n is the number of digits in this binary number, a represents the value on one of the bits of this binary number.

$$multiplier \times multmultiplicand = (-2A_{2n+1} + A_{2n} + A_{2n-1}) \times multmultiplicand \times 2^{2n} + \cdots + (-2A_3 + A_2 + A_1) \times multiplicand \times 2^2 + (-2A_1 + A_0 + A_{-1}) \times multmultiplicand \times 2^0 = multiplicand \times \left[\sum_{k=0}^{n} (-2A_{2n+1} + A_{2n} + A_{2n-1}) \times 2^{2k}\right] \tag{2}$$

After equivalent transformation of the above formula, it can be found that the number of terms in the polynomial expression has been reduced to half of the original. Starting from the LSB, the original binary number is grouped into threes (with an additional bit, A [-1], added to the lowest group, with a value of 0), with one bit overlapping between adjacent groups (the highest bit of the lower group overlaps with the lowest bit of the higher group), forming new polynomial factors. This is the improved Booth encoding method. In this way, we can reduce the number of terms in the original binary number by half, thereby reducing the number of partial products to half of the original.

This article uses the Booth encoding shown in the table 1, generating three control signals: X2, Nega, and Set0.

Table. 1 Encoded Signal

b2n+1	b2n	b2n-1	PPn	X2	Nega	set0
0	0	0	+0×A22n	0	0	1
0	0	1	+1×A22n	0	0	0
0	1	0	+1×A22n	0	0	0
0	1	1	+2×A22n	1	0	0
1	0	0	-2×A22n	1	1	0
1	0	1	-1×A22n	0	1	0
1	1	0	-1×A22n	0	1	0
1	1	1	-0×A22n	0	1	1

Their corresponding operations are shift, negation, and setting to 0 when the signals are active [2]. The logical expressions are as follows:

$$X2 = A_{2N} \odot A_{2N-1}$$
$$Nega = A_{2N+1} \tag{3}$$
$$Set0 = A_{2N+1} \cdot A_{2N} \cdot A_{2N-1} + \overline{A_{2N+1} \cdot A_{2N} \cdot A_{2N-1}}$$

2.2. Modified partial product array

When performing the inversion and increment operation on the partial products, it is necessary to invert and increment all the bits of the multiplicant. However, if this is done immediately after inverting the multiplicant, it will increase the delay of the partial product generation module. Therefore, this article adopts the method of delaying the increment operation until the 0 bit of the next partial product [1].

As the multiplier studied in this article deals with signed multiplication, the data in the partial products are in two's complement form. Before compressing the partial products, sign extension is required, which traditionally consumes considerable hardware resources. Therefore, this article applies special treatment to the sign bit of the partial products: the high bits of the first partial product PP1 are extended by 3 bits ($\bar{S}SS$), while the high bits of other partial products are extended to 2 bits ($1\bar{S}$)，where S represents the current sign bit of the partial product, $\bar{S}$ represents the negation of the sign bit. This treatment allows for more resource savings during the computation.

2.3. Wallace tree structure

2.3.1 Modified 4-2 compressor

4-2 compressor is an essential part of the Wallace tree compression structure used in this article. One drawback of this 4-2 compressor is that its carry signal generation requires a delay of 4 XOR gates [3]. This article adopts an optimized 4-2 compressor with a structure as shown in the figure 2.

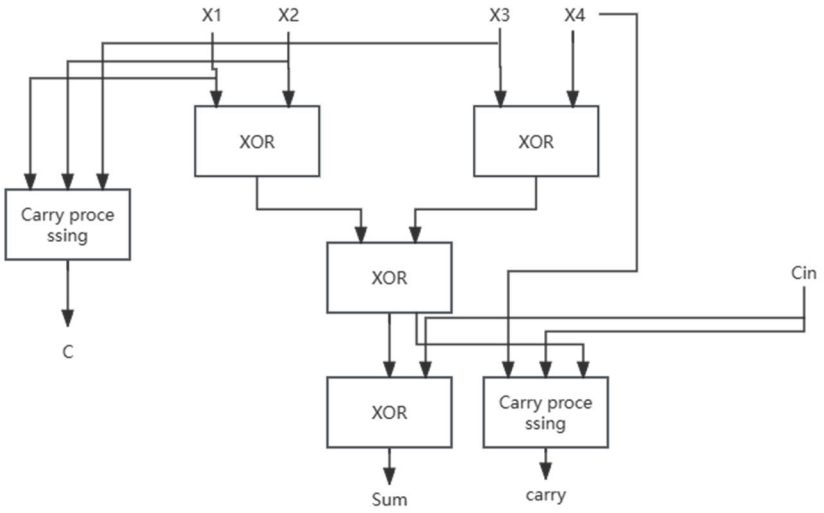

Fig. 2 Compressor structure (Photo/Picture credit: Original)

Its logical expression is:

$$C = (X_1 \cdot X_2) + (X_2 \cdot X_3) + (X_1 \cdot X_3)$$

$$Sum = X1 \oplus X2 \oplus X3 \oplus X4 \oplus Cin \tag{4}$$

$$carry = (X4 \cdot \overline{(X1 \oplus X2 \oplus X3 \oplus X4)}) + (Cin \cdot (X1 \oplus X2 \oplus X3 \oplus X4)$$

Compared to the traditional 4-2 compressor, the optimized 4-2 compressor reduces the delay of sum and carry values by one XOR gate.

2.3.2 Optimized Wallance tree compression structure

Due to 17 partial products in this multiplier, using a 3-2 compressor for summation would require 6 levels of CSA to compress them into 2 partial products. If a 4-2 compressor is used, 4 levels would suffice. Finally, the compressed partial products are added using a carry-lookahead adder to obtain the final result. To further conserve resources, a new Wallace tree structure is designed as shown, with p0~p16 being the 17 partial products generated by an improved Booth encoding circuit [4]. The Wallace tree structure is shown in figure 3.

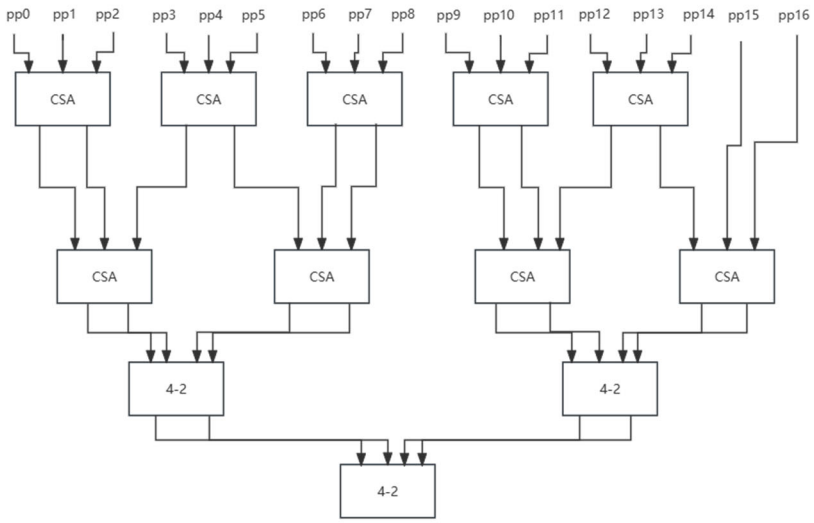

Fig. 3 Wallace tree structure (Photo/Picture credit: Original)

In summary, the tree structure used in this article has the least critical path delay and resource consumption compared to traditional tree structures. The number of compression units required, and the critical delay are shown in the table 2 [5].

Table.2 Comparison of three tree structures

Tree structure	The number of compression units consumed	Critical delay
Full use of 3-2 compressor	15 csa modules	12 xor
Full use of 4-2 compressor	8 4-2 compression modules	12 xor
The structure in this paper	9 csa modules and 3 4-2 compression modules	10 xor

2.4. carry-lookahead adder

The main drawback of a serial carry adder is the delay caused by waiting for the carry bit Ci to propagate one bit at a time when calculating each bit of S. To address this, we introduce the principle of a carry chain, a logic circuit that exists in an adder to propagate carry signals in parallel. Two signal bits, the carry generate bit (gi) and the carry propagate bit (pi), are introduced to make all bits' carry independent of lower bits' carry, allowing simultaneous generation of each bit's carry [6]. Thus, the delay in obtaining results is unrelated to the number of bits. The logical expressions are:

$$g_i = a_i \cdot b_i$$
$$p_i = a_i \oplus b_i$$
$$C_0 = Cin \tag{5}$$
$$C_{i+1} = g_i + p_i \cdot c_i$$
$$S_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$$

In theory, the design of a 64-bit adder can also be fully expanded according to the above derivation. However, due to the lengthy expression of ci, the fan-in and fan-out requirements for logic gates become excessive, making the circuit implementation complex. Therefore, a multi-level hierarchical implementation scheme, i.e., multiple groups jump carry, is adopted for the design of a 64-bit lookahead carry adder. The 64-bit input is decomposed into 16 groups of 4-bit lookahead carry adders, forming the second level of lookahead carry from the 16 groups obtained in the first level. The logic of lookahead carry is still used between groups, which can be further divided into 4 groups to form the third level [7]. Thus, the designed 64-bit lookahead carry adder is implemented with 3 levels of lookahead carry logic.

3. Experimental results and analysis

3.1. Test file code flow

This paper uses the ModelSim simulator to simulate the designed multiplier. ModelSim is a simulation software developed and tested by Mentor Graphics, which is widely used in industry. It can input test files and obtain simulation results. The simulation results can be output by waveform. The testbench is a module that tests the design and supports Verilog or VHDL language. We can write the testbench file to add incentive input to the design module. First, the testbench file is written by Verilog. Next, we introduce the code flow of the test file: First, we define the parameter period and the parameter *num_test*. The value of period is 5 and the value of *num_test* is 100000. *num_test* is defined to test 100000 signed random numbers. Of course, when defining variables, we initialize the *multiplier*, the *multiplier* and the *clock* to zero. Then set a random number of symbols, that is, the multiplier and the multiplier assignment [8]. An integer constant *i* is defined, its function is to count, count the number of calculations, whenever the multiplier and the multiplier is a random number of symbols, *i* plus 1. Then, the integer constants *num_correct* and *num_error* are defined, which represent the number of correct and wrong results, respectively. The *product _ ref* variable is defined to represent the reference result, which is the correct calculation result. Define *result*, take 0 when *product _ ref* is equal to product, otherwise take 1 [9]. Next, using the loop statement, when *result* =

0, the output is displayed correctly and *num_correct* is added to one, otherwise the output is displayed incorrectly and *num_error* is added to one. When the number of cycles to the upper limit, the output calculation results correct rate [10]. The following figure 4 is the test file code flow chart.

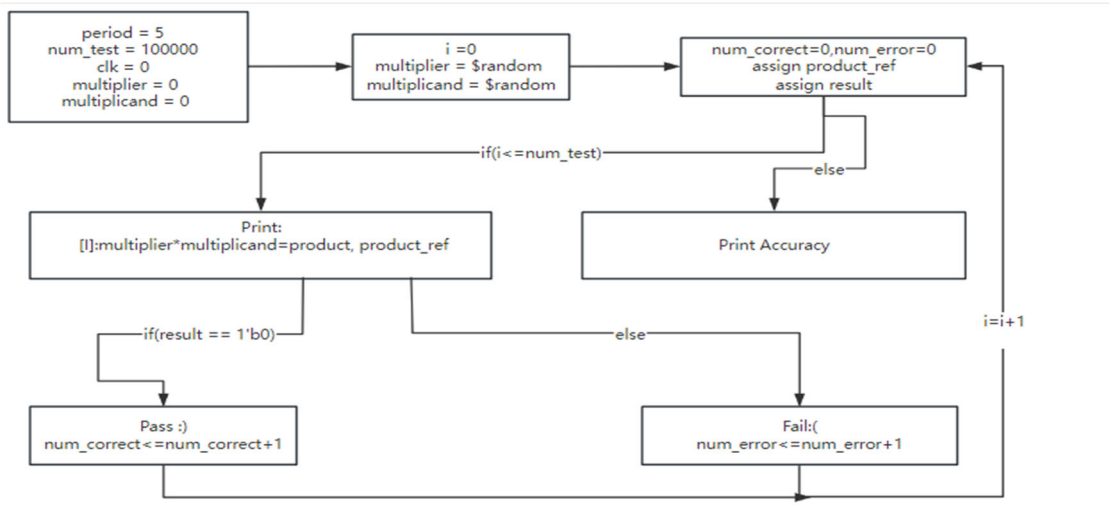

Fig. 4 Test file code flow chart (Photo/Picture credit: Original)

3.2. Simulation results and analysis

This paper set up 100,000 groups in the testbench to verify its correctness, check the output results of MadelSim, and the correct rate is 100 %. In the output waveform, we can observe that the input includes two *multiplier* and *multiplicand* signals with a bit width of 32, representing the values to be calculated for the two inputs, and there is also a clock signal. *Product* represents the calculation result of the output, which is a signal with a bit width of 64 bits.

On this basis, in order to verify the correctness of each calculation, we set up the *product _ ref* signal, which represents the reference result, that is, the correct calculation result; the *result* signal represents the difference between *product* and *product _ ref*. The figure shows that the output is always low, which means that the output result is correct. *num_correct* represents the number of correct results running to the current state, and *num_error* represents the number of error results running to the current state. The figure 5 shows that the output has been low, which means that the output result is correct.

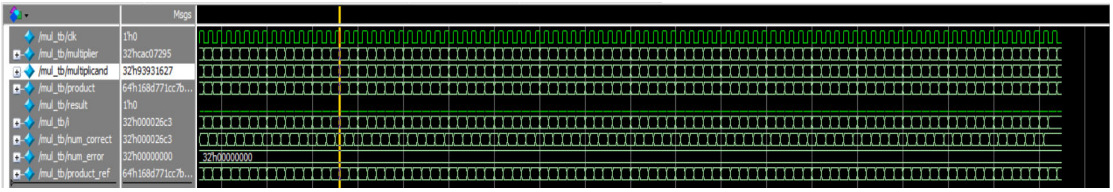

Fig. 5 Oscillogram (Photo/Picture credit: Original)

In the simulation result file, it can see the decimal output results of 100000 sets of data *product* and *product _ ref*. It is noted that the decimal numbers in the figure are the corresponding 32-bit binary numbers, and the random input data are positive and negative. After comparison, all passed the test. Through the above simulation test, this paper analyze that the multiplier can realize its function normally. And the result is shown in figure 6.

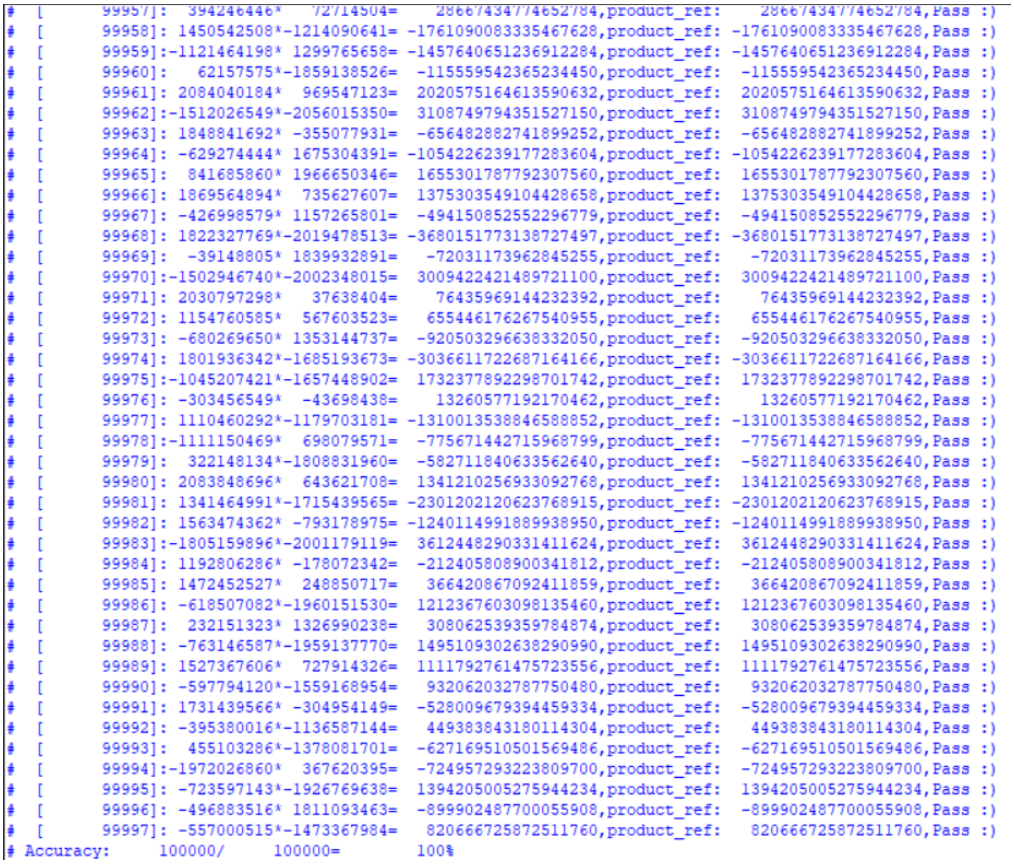

Fig. 6 Outcome (Photo/Picture credit: Original)

4. Conclusion

This study articulates the formulation of a 32-bit Booth multiplier, emphasizing the substantial enhancement in multiplier performance achievable through the strategic implementation of Booth encoding and Wallace tree methodologies. The efficacy and accuracy of this approach are substantiated via comprehensive theoretical scrutiny complemented by empirical validations. The multiplier configuration delineated herein is characterized by its superior velocity, minimized hardware exigency, and optimal energy thriftiness, thereby offering a refined and efficacious paradigm for multiplication tasks within computational infrastructures.

Regrettably, temporal limitations precluded the verification of circuit area and latency within the current study. Prospective investigations will delve deeper into the refinement and authentication of multiplier optimization strategies, including the integration of pipeline architectures, to address the burgeoning computational performance requisites and the expansive spectrum of real-world application necessities. Future research is poised to explore avant-garde multiplier frameworks, employing increasingly sophisticated optimization stratagems aimed at amplifying both performance and energy frugality. Moreover, ensuing studies will concentrate on harnessing emergent modalities such as parallel and distributed computation within multiplier design, aspiring to actualize exceedingly proficient computational systems.

Authors Contribution

All the authors contributed equally, and their names were listed in alphabetical order.

References

[1] Venkata Subbaiah M, Umamaheswara Reddy G. Design of Delay-Efficient Carry-Save Multiplier by Structural Decomposition of Conventional Carry-Save Multiplier. Intelligent Systems and Sustainable Computing: Proceedings of ICISSC 2021. Singapore: Springer Nature Singapore, 2022: 435-447.

[2] Jeevan B, Sivani K. A new high-speed multiplier based on carry-look-ahead adder and compressor. VLSI Design: Circuits, Systems and Applications: Select Proceedings of ICNETS2, Volume V. Springer Singapore, 2018: 69-78.

[3] Cooper A R. Parallel architecture modified Booth multiplier. IEE Proceedings G (Electronic Circuits and Systems). IET Digital Library, 1988, 135(3): 125-128.

[4] Wang G, Yi Q M. An optimized design of multiplier based on improved booth encoding and Wallace tree. Computer Applications and Software, 2016, 33(5): 13-16.

[5] Kang J Y, Gaudiot J L. A simple high-speed multiplier design. IEEE Transactions on computers, 2006, 55(10): 1253-1258.

[6] Rafiq A, Chaudhry S M. Design of an improved low-power and high-speed booth multiplier. Circuits, Systems, and Signal Processing, 2021, 40(11): 5500-5532.

[7] Upadhyay R M, Chauhan R K, Kumar M. Energy Efficient 4-2 and 5-2 Compressor for Low Power Computing. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal, 2023, 12(1): e30381-e30381.

[8] Alghamdi A, Gebali F. Performance analysis of 64-bit Carry Lookahead Adders using conventional and hierarchical structure styles. 2015 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM). IEEE, 2015: 80-83.

[9] Martinez B, Yang J, Bulat A, et al. Training binary neural networks with real-to-binary convolutions. arXiv preprint arXiv:2003.11535, 2020.

[10] Bhosale S, Raut K J, Deshmukh M, et al. Optimization of Partial Products in Modified Booth Multiplier. International Conference on Information and Communication Technology for Intelligent Systems. Singapore: Springer Nature Singapore, 2023: 267-277.