RESEARCH ARTICLE

International Journal of Computer Communication and Informatics

DOI: 10.34256/ijcci2221

Low Power and Efficient Re-Configurable Multiplier for Accelerator

N. Nikitha Reddy [a,] *, Gogula Subash [a], P. Hemaditya [a], Maran Ponnambalam [a]

[a] Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Chennai, Tamil Nadu 601103, India.

* Corresponding Author: nikitha2241@gmail.com

Received: 12-08-2022, Revised: 20-09-2022, Accepted: 28-10-2022, Published: 30-12-2022

Abstract: Deep learning is a rising topic at the edge of technology, with applications in many areas of our lives, including object detection, speech recognition, natural language processing, and more. Deep learning's advantages of high accuracy, speed, and flexibility are now being used in practically all major sciences and technologies. As a result, any efforts to improve the performance of related techniques are worthwhile. We always have a tendency to generate data faster than we can analyse, comprehend, transfer, and reconstruct it. Demanding data-intensive applications such as Big Data. Deep Learning, Machine Learning (ML), the Internet of Things (IoT), and high- speed computing are driving the demand for "accelerators" to offload work from general-purpose CPUs. An accelerator (a hardware device) works in tandem with the CPU server to improve data processing speed and performance. There are a variety of off-the-shelf accelerator architectures available, including GPU, ASIC, and FPGA architectures. So, this work focus on designing a multiplier unit for the accelerators. This increases the performance of DNN, reduced the area and increasing the training speed of the system.

Keywords: CNN, VHDL, Xilinx, FPGA, Multiplier, Acclerator.

1. Introduction

Ability of a computer or machine to perform a specific task based on the statistics of the past data is called artificial Intelligence (AI). Machine Learning (ML) and Deep Learning (DL) are the sub classification of AI [1]. Machine learning has become pervasive in numerous research sectors, commercial applications, and achieved satisfactory products over the last few years. Deep learning has accelerated the advancement of machine learning and artificial intelligence. Due to which the research on ML, DL and AI are exponentially increasing across various field all over the world [2]. Many recent artificial intelligence (AI) applications use deep neural networks (DNNs) as their core. The efficient hardware implementation of DNNs is critical as they proliferate further in real applications. The emergence of online learning has raised a great deal of interest in training DNNs on resource- constrained platforms. Unmanned vehicle, detection of chronic disease like cancer is some of the breakthrough applications, where DNN outperform humans.

As stated before, DNN is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input. Each layer is aimed at identifying features at various levels [3]. In image recognition, for example, where the input is initially in the form of pixels, the first layer recognizes low-level features like edges and curves. The first layer's output is fed into the second layer, which creates higher-level features like semi- circles and squares. A further layer assembles the previous layer's output into portions of familiar things, while a third layer detects the objects [4]. The network produces an activation map that symbolizes more and more complex features as we move through more layers. The filters become increasingly responsive to a greater region of the pixel space as you progress deeper into the network. Higher level layers boost discrimination-relevant elements of received inputs while suppressing irrelevant variations.

The Convolutional Neural Network (CNN, or ConvNet) is a class of deep neural networks, which is widely used for image/object recognition and classification. The size of the convolution neural network must be increased by adding more neural network layers to deliver more accurate results as well as real-time object detection, for example in applications like as robotics and self-driving cars. However, as more and different types of NN layers are developed, more complicated CNN structures and high-depth CNN models emerge. To train and test the resulting large-scale CNN, billions of operations and millions of parameters, as well as significant computer resources, are required [5]. For general purpose processors (GPP), such requirements provide a computational difficulty. As a result, hardware accelerators such as application specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), and graphics processing units (GPUs) have been used to boost the CNN's performance. GPU accelerators, on the other hand, consume a large amount of power. [6] In comparison to GPU-based accelerators, FPGA and ASIC hardware accelerators have relatively limited memory, I/O bandwidths, and computational resources. However, they can achieve at least moderate performance with lower power consumption.

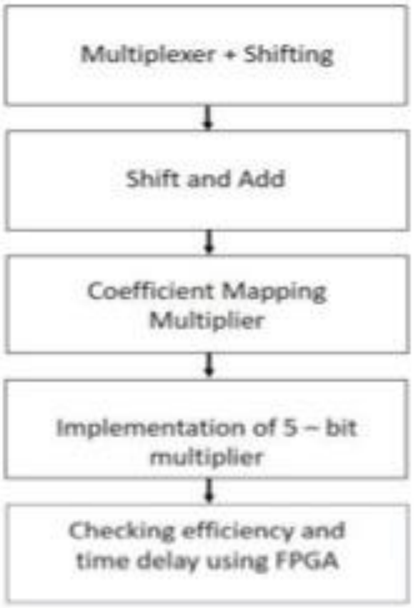
FPGAs have recently been found to perform well in inference and may speed up training [7]. GPUs and FPGAs provide significant speedups for deep learning computation, but they use a lot of power that can be handled by special-purpose accelerators. Reference article [8] presented a reconfigurable constant coefficient multipliers (RCCMs), which gives a superior alternative for reducing silicon space than using low- precision arithmetic. RCCMs use just adders, sub tractors, bit shifters, and multiplexers (MUXes) to multiply input values by a limited set of coefficients, allowing them to be substantially optimized for FPGAs. They suggested a set of RCCMs specifically designed for FPGA logic parts in order to maximize their efficiency.

The next unique training device translate the potential coefficient representations of the RCCMs to neural network weight parameter distributions [9] to reduce information loss due to quantization. This allows the RCCMs to be used in hardware retaining great accuracy. AlexNet, ResNet-18, and ResNet-50 networks are used to compare the benefits of these strategies. In comparison to typical 8-bit quantized networks, the resultant implementations save up to 50% in resources, resulting in considerable performance and power savings.

In this research, a novel design for an accelerator is described that can improve the efficiency of an FPGA processor. The architecture of this accelerator is made up of simple digital circuits like multiplexers, shift registers, multipliers, multiplexers, and adders. The main purpose is to build a multiplier unit for an accelerator in order to boost the processor's performance. VHDL behavioral modulation was used to create these circuits.

2. PROPOSED MODEL

Several algorithms for the multiplier have been developed for use in accelerators to speed up CNN network training time. The development of a multiplexer and shift registers is the first step in the proposed design. A multiplexer sends the desired left shifted / multiplied value to the output based on the selection line.


Multiplexer + Shifting
Shift and Add
Coefficient Mapping Multiplier
Implementation of 5 – bit multiplier
Checking efficiency and time delay using FPGA

Flow Chart

The desired coefficients are achieved by fundamentally shifting and adding the corresponding shifted intermediate outputs based on the multiplexer's selection input. A 4 bit value, I, for instance, needs to be multiplied by a 5 bit coefficient. The output is 5*I can be obtained by adding the result of intermediate output 4*I and I. Left shifting the given input data by 4 times gives the value of 4*I and adding this intermediate result with the I will give the desired coefficient value of 5*I. The kind of mux and the quantity of input bits determine the coefficient's value. The four stages of shifting with four different selection combinations are demonstrated in the example below. Additionally, the size of the multiplexer is determined by the quantity of input bits. Needs 4 to 1 mux for 4 bit input data. X bit input requires X to 1 multiplexer in general

I = 0001

I << 1 --> 0010 if s=00

I << 2 --> 0100 if s=01

I << 3 --> 1000 if s=10

I << 4 --> 0000 if s=11

The previous coefficient multiplier operation is reasonably easy if the coefficient has a single value. Additionally, the equation becomes too complicated if the CNN network demands greater numbers, such as 12305.

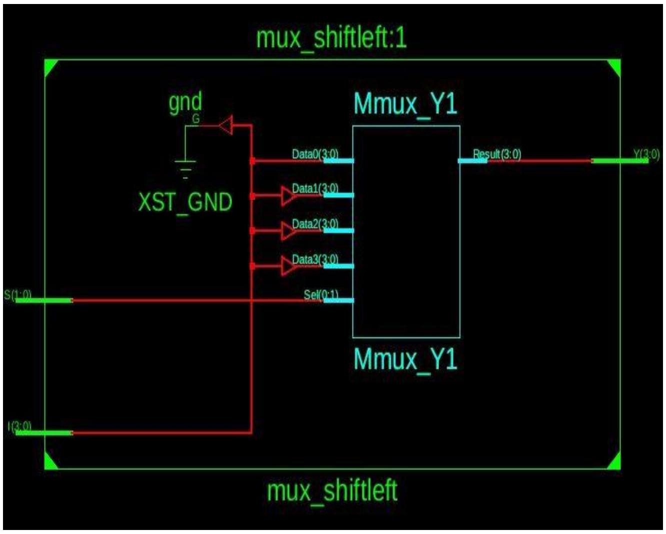


Figure 1: Mux and shift design using VHDL code in Xilinx

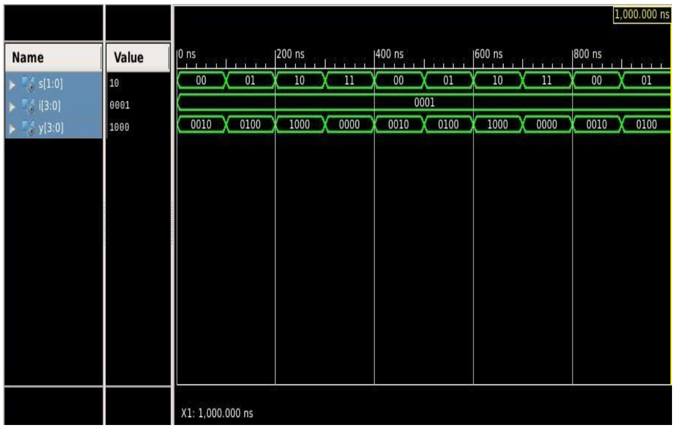


Figure 2: Output of the figure 1 design

The algorithm described in [9] was complex and took longer and more power than it was supposed to. A clear, less complex, and simpler shift add method was offered as a solution to this issue. In this algorithm, the required decimal number is first encoded in binary form. The specified input number is left-shifted by matching amounts in the bit locations where the 1 is situated. The desired coefficient multiplication is then obtained by adding the left-shifted values.

For example, consider the desired co-efficient as X=1100000000010001 (12305). The initial number is 0000000000000001. Now, we will be checking the at what bit places 1's are there. For the given example, 1's are placed in 4, 14, 15.

So, we will be shifting the binary number 1.

A = X<<6 --> 0000000000010000

B = X<<7 --> 0100000000000000

C = X<<8 --> 1000000000000000

Y = X+A+B+C--> 1100000000010001

The above pseudo code is implemented in Xilinx as shown in figure 3.

Table 1: 5 X 5 Coefficient Weighted Matrix

	2^4	2^3	2^2	2^1	2^0
2^4	256	128	64	32	16
2^3	128	64	32	16	8
2^2	64	32	16	8	4
2^1	32	16	8	4	2
2^0	16	8	4	2	1

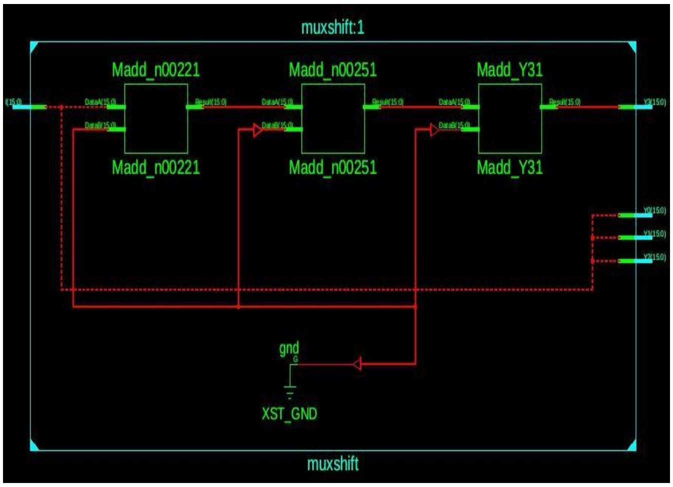
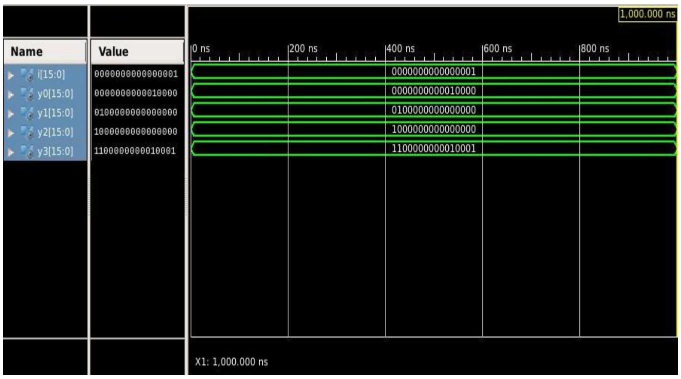
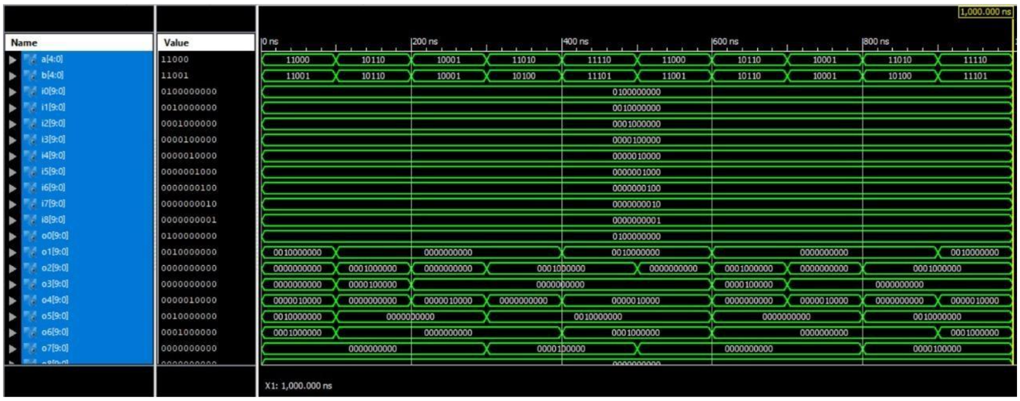


Figure 3: Shift and add to get the desired coefficient.



```
Name        Value
i[15:0]     0000000000000001
y0[15:0]    0000000000010000
y1[15:0]    0100000000000000
y2[15:0]    1000000000000000
y3[15:0]    1100000000010001
X1: 1,000.000 ns
```

Figure 4: Output of the implemented design shown above in figure 3.



```
Name     Value
a[4:0]   11000
b[4:0]   11001
i0[9:0]  0100000000
i1[9:0]  0010000000
i2[9:0]  0001000000
i3[9:0]  0000100000
i4[9:0]  0000010000
i5[9:0]  0000001000
i6[9:0]  0000000100
i7[9:0]  0000000010
i8[9:0]  0000000001
o0[9:0]  0100000000
o1[9:0]  0010000000
o2[9:0]  0000000000
o3[9:0]  0000000000
o4[9:0]  0000010000
o5[9:0]  0010000000
o6[9:0]  0001000000
o7[9:0]  0000000000
X1: 1,000.000 ns
```

Figure 5a. Simulation results of the proposed 5 bit CMM.

Compared to the algorithm discussed in reference paper [9], this one is simpler and faster. Even with this strategy, the shifting operation varies depending on the coefficient.

The goal of this work is to create a generic method that is easy to use, quick, and energy-efficient for multiplying any complex coefficients. The base values' matrix representation serves as the foundation for the suggested Coefficient Matrix multiplier.

For example, if the given multiplication is 25*25, at first convert the decimal numbers to binary numbers and represent the binary number in the rows and columns of the above specified matrix as shown in figure 5. The cells where the rows having 1's and columns having 1's are intersecting are the intermediate product terms as highlighted in Figure 5 a&b. The weightage in the highlighted cells is added row wise and then the sum of the values is added in the last column. This will lead to a coefficient of 625. This approach can be extended to any size of the matrix of size N X N, with an input bit length of N.

Input/Input T		1	1	0	0	1	Total
		2^4	2^3	2^2	2^1	2^0	
1	2^4	**256**	**128**	64	32	**16**	400
1	2^3	**128**	**64**	32	16	**8**	200
0	2^2	64	32	16	8	4	0
0	2^1	32	16	8	4	2	0
1	2^0	**16**	**8**	4	2	**1**	25
				Total		=	**625**

When compared to existing algorithms, the proposed CMM algorithm increases computational speed and efficiency while decreasing memory cell utilisation. The main issue with this proposed CMM is the large number of memory cells required to store the length of the input or the coefficient multiplier. The above Coefficient matrix shows that only 9 weighted coefficients are repeated in the remaining cells. To eliminate redundancy, only 9 memory cells are used, with weighted coefficients of 256, 128, 64, 32, 16, 8, 4, 2, 1. The values in the 9 memory cells are then reused based on the intersecting cell positions to reduce memory usage.

3. Simulation

The proposed CMM algorithm with is modeled and verified in the Xilinx using behavioral modeling on the XC6SLX75L- FGG676 device from "SPARTAN 6 lower power" FPGA family. Several test cases were used to determine whether or not the output generated was correct. Consider the numbers 11000 and 11001 as inputs. When utilising the coefficient matrix multiplier procedure, add all of the numbers 256+128, 128+64, and 16+8. The maximum output is 600. This number's binary representation is 1001011000. That's the same as the xilinx simulation results. Inputs are represented in binary as 24 and 25. The result of multiplying 24 and 25 is 600.

4. Analysis of the Proposed Algorithm

This section discusses the Xilinx ISE design summary report. The targeted device was from Spartan 6, with part number Xc6slx75l. Using the Xilinx XC6SLX75L-FGG676 chip from the "SPARTAN 6 lower power" line of FPGA boards, the proposed model was compared to other models presented in various algorithms described in research papers. This basic board was chosen because it is the most cost-effective and widely used. On the same target device, this section compares the performance metrics of the proposed RCCM with the shift and add algorithm and the CMM method.

Table 2 summarises and compares the device utilisation summary, which compares the total number of available resources and the total number of resources used in the above mentioned targeted device upon implementing the proposed RCCM. Table 3 also lists the number of slices, input and output buffers, and look-up tables used in implementing the proposed RCCM method with the existing methods.

Table 4 details the total power used in the circuit at different temperature conditions. An Xpower Analyzer is used to calculate power. Similarly, Table 5 summarises and compares the number of fan-outs from each of the buffers, look-up tables (LUT), multiplexers, etc. with other methods. It also includes the gate delays and net delay from each of the combinational blocks and calculates the overall critical delay involved in the implementation of the proposed RCCM.

In the simulation described above, coefficient matrix multiplier is performed using two 5-bit values, and in the reference paper [10], they have simulated two 4-bit numbers for array and booth multiplier. They have achieved a delay of 23.856 and 30.798ns respectively for array and booth multipliers as shown in table 6. The proposed algorithm achieved a delay of 15.205ns. Even though it has an extra bit, the proposed algorithm coefficient matrix multiplier has a significantly lower time delay than these other algorithms.

Table 2: Device utilisation summary

Logic Utilization	Used	Available
Number of slice LUTs	80	46648
Number used as logic	80	46648
No, of LUT Flipflop pairs used	80	80
Number with an Unused flip flop pairs	0	80
Number with an unused LUT	0	80
No, of fully used LUT-FF pairs	0	80
No, of unique Control sets	0	
No, of Bounded IOBs	360	408

Table 3: Comparison of device utilization between shift and add and CMM

Logic Utilization	Used		Available		Utilization	
	CMM	Shift and ADD	CMM	Shift and ADD	CM M	Shift and ADD
No of Slices	80	32	46648	46648	0%	0%
No of fully used LUT-FF pairs	0	0	80	32	0%	0%
No of bonded IOBs	360	80	408	408	88%	19%

Table 4: Comparison of power between shift and add and CMM

On-chip	Power(W)		Used		Available		Utilization(%)	
	CMM	Shift and ADD	CMM	Shift and ADD	CMM	Shift and ADD	CMM	Shift and ADD
Logic	0.000	0.000	16	46	46648	46648	0	0
Signals	0.000	0.000	45	85	---	---	---	---
IOs	0.000	0.000	80	360	408	408	20	88
Leakage	0.046	0.046						
Total	0.046	0.046						

Table 5: Comparison of Delay between Shift and add and CMM

Cell:in -> out	Fan-out		Gate Delay		Net delay	
	CMM	Shift and ADD	CMM	Shift and ADD	CMM	Shift and ADD
IBUF:I->O	9	9	1.594	1.594	1.332	1.332
LUT2:I0->O	3	1	0.454	0.454	1.697	0.818
LUT6:I0->O	2	1	0.454	0.430	0.869	0.000
LUT3:I2->O	1	1	0.430	0.370	0.818	0.000
LUT4:I3->O	1	1	0.430	0.015	0.000	0.000
MUXCY:S->O	1	1	0.370	0.015	0.000	0.000
MUXCY:CI->O	1	1	0.015	0.015	0.000	0.000
MUXCY:CI->O	1	1	0.015	0.015	0.000	0.000
XORCY:CI->O	1	1	0.393	0.393	0.818	0.000
LUT2:I1->O	1	1	0.430	0.430	0.000	0.000
MUXCY:S->O	0	0	0.370	0.370	0.000	0.000

XORCY:CI->O	1	1	0.393	0.393	0.783	0.783
OBUF:I->O			3.540	3.540		
Total	CMM= 15.205ns (8.888ns Logic, 6.317ns route) Shift and ADD= 9.850ns (6.916ns Logic, 2.934ns route)					

Table 6 : Comparison of multipliers

	Array	**Booth**	**Shift and Add**	**CMM**
No of bits	4 bits	4 bits	16 bits	5 bits
Delay (ns)	23.856	30.798	9.850	15.205
I/O Buffers	32	33	80	360
Bels(Area)	123	458	64	138
Power	0.081	0.014	0.046	0.046

5. Conclusion

In this study, many techniques for multiplying diverse numbers were examined, and a brand-new algorithm called coefficient matrix multiplier was proposed. The procedure discussed here multiplies the integers quickly and simply using a matrix. For multiplying two 5-bit values, a new model has been developed. After deploying the code in FPGA board, we have obtained a delay of 15.205 ns and power of 0.046 W. The coefficient matrix algorithm is compared to the shift and add procedure and other multipliers like array and booth. The coefficient matrix multiplier differs from other multipliers in that once the code is written, it can be used to multiply two different numbers without requiring revision, whereas the shift and add algorithm requires revision for each and every multiplication. We have determined that coefficient matrix multiplier is the best option and is the easiest to use after examining the delay, power, and area.

References

[1] Y. Bengio, Learning deep architectures for AI, Foundations and trends® in Machine Learning, 2(1) (2009) 1– 127.

[2] J. Schmidhuber, Deep learning in neural networks: An overview, Neural networks, 61(2015) 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[3] Vagisha Gupta, Shelly Sachdeva and Neha Dohare, Deep similarity learning for disease prediction, Trends in Deep Learning Methodologies, (2021) 183-206. https://doi.org/10.1016/B978-0-12-822226-3.00008-8

[4] Y. LeCun, Y. Bengio, and G. Hinton, Deep learning, Nature, 521 (7553) (2015) 436. https://doi.org/10.1038/nature14539

[5] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, Project Adam: Building an efficient and scalable deep learning training system, in OSDI, 14 (2014) 571–582.

[6] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pages 1–4, Aug 2016. https://doi.org/10.1109/FPL.2016.7577314

[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, *FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 161-170 (2015). https://doi.org/10.1145/2684746.2689060

[8] J. Faraone et al., "AddNet: Deep Neural Networks Using FPGA- Optimized Multipliers," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 28(1) (2020) 115-128. https://doi.org/10.1109/TVLSI.2019.2939429

[9] M. Courbariaux and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1" in CoRR, 2016, [online] Available: http://arxiv.org/abs/1602.02830.

[10] M. Aravind Kumar, O. Ranga Rao, M. Dileep, C V Pradeep Kumar Reddy, K.P. Mani Performance Evaluation of Different Multipliers in VLSI using VHDL, in International Journal of Advanced Research in Computer and Communication Engineering 5(3) (2016).

Funding

No funding was received for conducting this study.

Conflict of interest

The Authors have no conflicts of interest to declare that they are relevant to the content of this article.

About The License

© The Author(s) 2022. The text of this article is open access and licensed under a Creative Commons Attribution 4.0 International License