西 南 交 通 大 学 学 报

第 56 卷 第 3 期
2021 年 6 月

JOURNAL OF SOUTHWEST JIAOTONG UNIVERSITY

Vol. 56 No. 3
June 2021

**ISSN: 0258-2724** **DOI：10.35741/issn.0258-2724.56.3.11**

Research article

Electrical and Electronic Engineering

# NEW CONCEPT OF UNIVERSAL BINARY MULTIPLICATION AND ITS IMPLEMENTATION ON FPGA

# 通用二进制乘法的新概念及其在现场可编程门阵列上的实现

**Sarifuddin Madenda*, Suryadi Harmanto, Astie Darmayantie**
Computer Engineering and Information Technology, Gunadarma University
Jl. Margonda Raya. No. 100, Depok – Jawa Barat, Indonesia, sarif@staff.gunadarma.ac.id,

*Received: March 13, 2021 ▪ Review: April 15, 2021 ▪ Accepted: May 14, 2021 ▪ Published: June 30, 2021*

**Abstract**
This paper proposes the new improvements of signed binary multiplication equation, signed multiplier, and universal multiplier. The proposed multipliers have low complexity algorithms and are easy to implement into software and hardware. Both signed, and universal multipliers are embedded into FPGA by optimizing the use of LUTs (6-LUT and 5-LUT), carry chain Carry4, and fast carry logics: MUXCYs and XORCYs.Each one is implemented as a serial-parallel multiplier and parallel multiplier. The signed multiplier executes four types of multiplication, i.e., between two operands that each one can be a signed positive (SPN) or signed negative numbers (SNN). The universal multiplier can handle all (nine) types of multiplication, where each operand can be as unsigned(USN), signed positive, and signed negative numbers. For 8x8 bits, signed serial-parallel and signed parallel multipliers occupy19 LUTs and 58 LUTs with a logic time delay of 0.769 ns and 3.600 ns. Besides, for 8x8 bits, serial-parallel and parallel universal multipliers inhabit 21 LUTs and 60 LUTs with a logic time delay of 0.831ns and 3.677 ns, successively.

**Keywords:** Signed Multiplication Algorithm, Signed and Universal Multipliers, FPGA, Fast Carry Logic, Carry Chain

**摘要** 本文提出了有符号二进制乘法方程、有符号乘法器和通用乘法器的新改进。所提出的乘法器算法复杂度低，易于实现到软件和硬件中。通过优化查找表（6-查找表和 5-查找表）、进位链进位 4 和快速进位逻辑：进位链的多工器的使用，有符号和通用乘法器都嵌入到现场可编程门阵列中和进位链的唯一“或”。每一个都被实现为串并乘法器和并行乘法器。有符号乘法器执行四种乘法，即在两个操作数之间，每个操作数可以是有符号正数或有符号负数。通用乘法器可以处理

所有（九种）乘法类型，其中每个操作数可以是无符号数、有符号正数和有符号负数。对于 8x8 位，有符号串行-并行和有符号并行乘法器占用 19 个查找表和 58 个查找表，逻辑时间延迟为 0.769 纳秒和 3.600 纳秒。此外，对于 8x8 位，串并和并行通用乘法器依次占据 21 个查找表和 60 个查找表，逻辑时间延迟分别为 0.831 纳秒和 3.677 纳秒。

**关键词:** 有符号乘法算法、有符号和通用乘法器、现场可编程门阵列、快速进位逻辑、进位链。

## I. INTRODUCTION

Multiplication is one of the arithmetic operations that play an important role in various computational processes carried out by computers. Computational speed depends on the complexity of a multiplication algorithm, and the number of multiplication operations performed to solve a given problem. For example, all algorithms related to multimedia data processing such as signal, image, and video processing require many multiplication operations. A huge amount of multimedia data processing algorithms used in Information and Communication Technology must be implemented into the System on Chip (SoC). It is intended so that multimedia data can be processed and communicated in real-time.

Currently, automation technology related to signal, image, and video analysis and processing based on artificial intelligence continue to be developed. The use of Convolutional Neural Network (CNN), one of the artificial intelligence methods (AI), in the field of image processing is constantly expanding: biometrics recognition for personal identification [1], image recognition [2] [3], autonomous vehicles [4], medical diagnostics [5], and so on. In [6] proposes implementing CNN architecture in FPGA based on mapping and pipeline implementation methods on all its layers. [7] proposed a convolutional method using Sobel kernels in the convolutional layer of CNN and its hardware implementation on FPGA . Other implementation methods of acceleration in the deep learning network on FPGA are discussed in [8].

The multiplication and division operations are a major part of data processing algorithms in AI: Machine learning, CNN, and Deep learning. In [9] proposes the Stochastic Computing Multiplier method that is applied to the implementation of Deep Convolutional Neural Networks, where the embedment of each perceptron in the NN layer performs the same number of mathematical operations (additions, products, and threshold functions) [10]. Xilinx [11] is also developing an implementation model for the FPGA Acceleration of Matrix Multiplication for artificial neural networks. However, implementing all AI algorithms into the SoC (FPGA and ASIC) is often constrained by implementing multiplication and division operations. The constraint in question concerns the amount of space occupation on the integrated circuits (IC) related to production costs and the complexity of its implementation methods. The multiplication model depends on the binary data types used for multiplicand *A* and multiplier *B*. The data types of these two operands and their product are shown in Figure 1. Signed binary numbers (SN) mean that both positive and negative numbers may be represented. The most significant bit (MSB) indicates the sign, where bit sign “0” for signed positive number (*SPN*) and "1" for a signed negative number (*SNN*). Unsigned binary numbers (*USN*) refer to the numbers that only have a positive value without a sign bit. So, referring to the data types of binary numbers, a multiplier is expected to process nine multiplication functions, as shown in Figure 1 and Table 1.

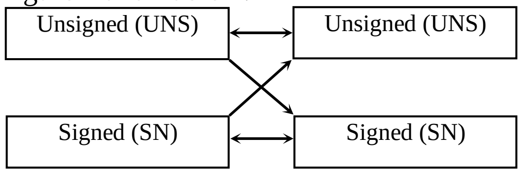


Figure 1. Type of binary multiplication

Table 1.
Nine types of binary multiplication

| Type | Multiplicand *B* | Multiplier *A* | Product *Y* = *B*×*A* |
|---|---|---|---|
| 1 | Unsigned (*USN*) | Unsigned (*USN*) | Unsigned (*USN*) |
| 2 | Unsigned (*USN*) | Signed Negative (*SNN*) | Signed Negative (*SNN*) |
| 3 | Unsigned (*USN*) | Signed Positive (*SPN*) | Signed Positive (*SPN*) |
| 4 | Signed Negative (*SNN*) | Unsigned (*USN*) | Signed Negative (*SNN*) |
| 5 | Signed Positive | Unsigned (*USN*) | Signed Positive (*SPN*) |

| | | | |
|---|---|---|---|
| | (*SPN*) | | |
| 6 | Signed Negative (*SNN*) | Signed Negative (*SNN*) | Signed Positive (*SPN*) |
| 7 | Signed Negative (*SNN*) | Signed Positive (*SPN*) | Signed Negative (*SNN*) |
| 8 | Signed Positive (*SPN*) | Signed Negative (*SNN*) | Signed Negative (*SNN*) |
| 9 | Signed Positive (*SPN*) | Signed Positive (*SPN*) | Signed Positive (*SPN*) |

A universal multiplier that can do all nine multiplications has been proposed in [12]. These multiplication processes are carried out following the rules of equation (1) and Figure 2. Both operands A and B must be checked first; if the value is negative, it will be converted to an absolute value, and then the multiplication process is carried out. The hardware implementation of this multiplier becomes more complex, increases time delay, and requires more electronic components so that production costs become expensive.

$$Y = \begin{cases} B * A & if\ A\ and\ B\ positives \\ |B| * A & if\ A\ positive\ and\ B\ negative \\ B * |A| & if\ A\ negative\ and\ B\ positive \\ |B| * |A| & if\ A\ and\ B\ negatives \\ sign\ bit = A_{sign} \oplus B_{sign} \end{cases} \quad (1)$$

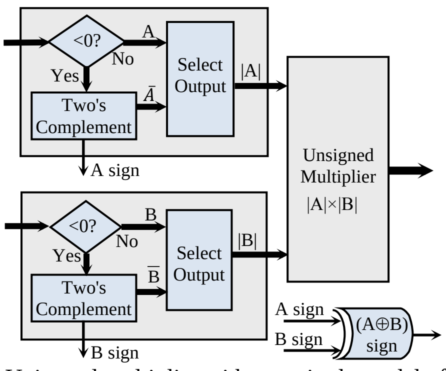


Figure 2. Universal multiplier with magnitude model of two signed numbers

Signed binary multiplication is one of the multiplications that is still a part of research topics, including developing the implementation methods with low complexity, low-cost hardware implementation, low-power consumption, and faster. Signed binary multiplication was introduced by Baugh-Wooley and then modified into two's complement multiplication, also known as modified Baugh-Wooley multiplication [13]. In [14], a fast implementation on FPGA of two's complement serial-parallel multiplier based on the modified Booth algorithm was proposed. Also, an efficient two's complement multiplier has been developed in [15]. Many other researchers focused on developing multipliers based on both Baugh-Wooley and two's complement multiplications using constant coefficient methods for the specific applications, especially how to implement them into FPGA or ASIC using minimum resources. This multiplication model is necessary for implementing exponential, logarithmic and trigonometric operations with constant multiplier values. In [16] introduced a multiplication model by an integer constant using minimum adders. A constant-coefficient multiplier (CCM) and its implementation methods into Xilinx FPGAwas developed in [17]. Some optimizations of CCM hardware implementation have been proposed in [18], [19], [20]. Some multiple constant multiplication (MCM) methods have been introduced and developed in [21], [22], [23], [24], [25] [26], [27], [28], and in [29] used a Look-Up Tables (LUTs) architecture to implement Constant Coefficient Multiplier into FPGA.

## II. State of The Art

### A. Binary Numbers and Two's Complement

Consider two unsigned binary numbers $A = \{a_{k-1}, a_{k-2}, \ldots, a_1, a_0\}$ and $B = \{b_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$, their decimal values can be expressed as shown in equation (2), where K and $n$ respectively are the number of bits, $a_i, b_j \in \{0,1\}$ $a_i$ is the $i^{th}$ magnitude bit of $A$ and $b_j$ is the $j^{th}$ magnitude bit of $B$. In the multiplication process, the conversion of unsigned binary numbers to signed numbers and from positive to a negative value, vice-versa, is necessary. For example, if $A$ is an unsigned binary number always positive, converting $A$ to a signed negative number is done through the two's complement process, as shown in equation (3). Because $A$ does not have a sign bit, then one bit indicating a negative sign "-1" must be added to the bit position of $2^n$, and then the one's complement process is done by inverting the $a_i$ bits into $\bar{a}_i$ and adding "1" to the LSB position of $a_0$. The sign bit "-1" can be replaced by a bit symbol," $\widehat{1}$ ", which means a borrowed bit has a negative value. Take an example, for unsigned $A = 1010_2$ or its decimal numbers is $A = 2^3+2^1 = 10_{10}$.Referring to equation (3), the two's complement of $A$ is $-A$ =

$\overset{\frown}{1}0101_2 + 1_2 = \overset{\frown}{1}0110_2$ or in decimal is $-2^4+2^2+2^1 = -10_{10}$.

$$A = \sum_{i=0}^{k-1} a_i 2^i \quad \text{and} \quad B = \sum_{j=0}^{n-1} b_j 2^j \qquad (2)$$

$$-A = -1.2^n + \left( \sum_{i=0}^{n-1} \bar{a}_i 2^i \right) + 2^0 \quad \text{or}$$

$$-A = \overset{\frown}{1}.2^n + \left( \sum_{i=0}^{n-1} \bar{a}_i 2^i \right) + 2^0 \qquad (3)$$

Furthermore, both can be represented if $A$ and $B$ are signed numbers, as shown in equation (4). $A = \{ \overset{\frown}{a}_{k-1}, a_{k-2}, \ldots, a_1, a_0\}$ and $B = \{ \overset{\frown}{b}_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$, where bits $\overset{\frown}{a}_{k-1}$ and $\overset{\frown}{b}_{n-1}$ are sign bits, or we call as MSB borrowed bits which mean $\overset{\frown}{a}_{k-1} = -1a_{k-1}$ and $\overset{\frown}{b}_{n-1} = -1b_{n-1}$. If $a_{k-1}$ = "0", $A$ has a positive value, and $a_i$ represents the magnitude bit. Conversely, if $a_{k-1}$ = "1", then $A$ is negative, and $a_i$ indicates its two's complement bit. The same thing applies to $B$.

$$A = \overset{\frown}{a}_{k-1} 2^{k-1} + \sum_{i=0}^{k-2} a_i 2^i \quad \text{and}$$

$$B = \overset{\frown}{b}_{n-1} 2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j \qquad (4)$$

$$-A = \overset{\frown}{\bar{a}}_{k-1} 2^{k-1} + \sum_{i=0}^{k-2} \bar{a}_i 2^i + 2^0 \quad \text{and}$$

$$-B = \overset{\frown}{\bar{b}}_{n-1} 2^{n-1} + \sum_{j=0}^{n-2} \bar{b}_j 2^j + 2^0 \qquad (5)$$

Next, the conversion of positive to negative binary numbers and vice versa is given by equation (5). Example, for signed binary numbers $A = \overset{\frown}{1}111_2$ or in decimal $A = -2^3+2^2+2^1+2^0 = -1_{10}$, then its two's complement is $-A = \overset{\frown}{0}000_2 + 1_2 = \overset{\frown}{0}001_2$ or $-A = 2^0 = 1_{10}$. Another example, if $A = \overset{\frown}{0}111_2$ or $A = 2^2+2^1+2^0 = 7_{10}$, then its two's complement is $-A = \overset{\frown}{1}000_2 + 1_2 = \overset{\frown}{1}001_2$ or $-A = -2^3+2^0 = -7_{10}$.

**B. Signed Baugh-Wooley's Multipliers**

Baugh and Wooley have proposed a signed binary multiplication SN×SN [13] and then modified and implemented as two's complement multiplier. These multiplications have been formulated according to equations 2-5, which are given respectively by equations 6 and 7 (for $k$=$n$). Figure 3 and Figure 4 show the shift-and-add and array structure of Baugh-Wooley's signed multiplier and two's complement multiplier for $n$ = 4. Note that the two's complement multiplier(equation 9) has a limitation. It only acts as an SNN×SNN multiplier, while the Baugh-Wooley's signed multiplier (equation 8) can process four types of multiplication: SPN×SPN, SPN×SNN, SNN×SPN, and SNN×SNN. However, its array structure (Figure 3) requires three additional full adders (FAs in gray color), so there is an increase in cost and time delay. It will significantly impact when used in the algorithms that require tens or hundreds of multipliers, such as in CNN.

## III. Proposed Universal Multiplier

The idea of developing a universal multiplier arose after considering all algorithms of audio, image, and video data processing that require all types of multiplication in table 1, as well as the structural similarity of these multiplications. In this section, a new modification of Baugh-Wooley's multiplication will be outlined. This proposed multiplication equation is expressed mathematically and easily implemented into software algorithms and integrated into hardware FPGA. Furthermore, a universal multiplier design and its hardware implementation method are proposed by referring to this proposed multiplication equation.

SN×SN:
$$Y = -2^{2n-1} + (a_{n-1}b_{n-1} + \bar{a}_{n-1} + \bar{b}_{n-1})2^{2n-2} + (a_{n-1} + b_{n-1})2^{n-1} + \sum_{i=0}^{n-2} \bar{a}_i b_{n-1} 2^{i+n-1} + \sum_{j=0}^{n-2} a_{n-1} \bar{b}_j 2^{j+N-1} + \sum_{j=0}^{n-2} \sum_{i=0}^{n-2} a_i b_j 2^{i+j} \qquad (6)$$

2's Comp.:
$$Y = 2^{2n-1} + a_{n-1}b_{n-1} 2^{2n-2} + 2^n + \sum_{i=0}^{n-2} \overline{b_{N-1}a_i}\, 2^{i+n-1} + \sum_{j=0}^{n-2} \overline{b_j a_{n-1}}\, 2^{j+n-1} + \sum_{j=0}^{n-2} \sum_{i=0}^{n-2} a_i b_j 2^{i+j} \qquad (7)$$

(a)
$$\begin{array}{llllll}
A(k=4) & & a_3 & a_2 & a_1 & a_0 \\
\times B(n=4) & & b_3 & b_2 & b_1 & b_0 \\
\hline
\bar{b}_0 a_3 \ b_0 a_2 \ b_0 a_1 \ b_0 a_0 & & & & & \\
\bar{b}_1 a_3 \ b_1 a_2 \ b_1 a_1 \ b_1 a_0 & & & & & \\
\bar{b}_2 a_3 \ b_2 a_2 \ b_2 a_1 \ b_2 a_0 & & & & & \\
-1 \ \ b_3 a_3 \ \ b_3 \bar{a}_2 b_3 \bar{a}_1 b_3 \bar{a}_0 & & & & & \\
\bar{a}_3 a_3 & & & & & \\
\bar{b}_3 b_3 & & & + & & \\
\hline
Y_7 Y_6 \quad Y_5 \quad Y_4 \quad Y_3 & & Y_2 & Y_1 & Y_0 &
\end{array}$$

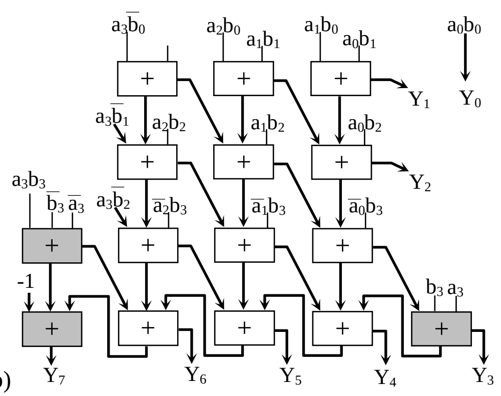
(b)


Figure 3. Shift-and-add and array structure of Baugh-Wooley's signed multiplier

(a)
$$\begin{array}{llllll}
A(k=4) & & a_3 & a_2 & a_1 & a_0 \\
\times B(n=4) & & b_3 & b_2 & b_1 & b_0 \\
\hline
1\overline{b_0 a_3}\, b_0 a_2 \ b_0 a_1 \ b_0 a_0 & & & & & \\
\overline{b_1 a_3}\, b_1 a_2 \ b_1 a_1 \ b_1 a_0 & & & & & \\
\overline{b_2 a_3}\, b_2 a_2 \ b_2 a_1 \ b_2 a_0 & & & & & \\
1 \ \ b_3 a_3 \ \ \overline{b_3 a_2 b_3 a_1 b_3 a_0} + & & & & & \\
\hline
Y_7 Y_6 \quad Y_5 \ Y_4 \quad Y_3 & & Y_2 & Y_1 & Y_0 &
\end{array}$$

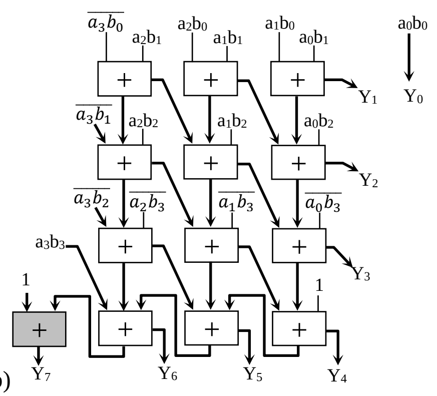
(b)


Figure 4. Modified Baugh-Wooley or two's complement multiplier

## A. Mathematical Approach

Based on equation (4), the signed multiplication of $Y = B \times A$ is shown by equation (8) and then equation (9), where $\hat{a}_{k-1}$ and $\hat{b}_{n-1}$ are respectively the sign bits of $A$ and $B$. Furthermore, referred to the equation (5), the second and third parts of equation (9) can be written in the form of two's complement as presented in equations (10) and (11). By inserting both into equation (9), then equation (12) is obtained. The first part of this equation is $(\hat{a}_{k-1}\hat{b}_{n-1} + \hat{a}_{k-1} + \hat{b}_{n-1})$can be simplified using the logical process in table 2. This part can be represented by two bits at position $2^{n+k-1}$and$2^{n+k-2}$. If one or both $\hat{a}_{k-1}$ and $\hat{b}_{n-1}$ equal "-1", then $(\hat{a}_{k-1}\hat{b}_{n-1}+\hat{a}_{k-1}+\hat{b}_{n-1})$ = "-1" and the bits at $2^{n+k-1}$ = "-1" and $2^{n+k-2}$ = "1". Otherwise $(\hat{a}_{k-1}\hat{b}_{n-1}+\hat{a}_{k-1}+\hat{b}_{n-1})$ = "0", if $\hat{a}_{k-1} = \hat{b}_{n-1}$ = "0", and the bits at $2^{n+k-1}$ = "0" and $2^{n+k-2}$ = "0". Thus by using the logical operation "OR" (denoted by ||), $(\hat{a}_{k-1}\hat{b}_{n-1}+\hat{a}_{k-1}+\hat{b}_{n-1})2^{n+k-1}$ can be replaced by $(\hat{a}_{k-1}||\hat{b}_{n-1})2^{n+k-1}$ + $(a_{k-1}||b_{n-1})2^{n+k-2}$ as given in equation (13).

Table 2.
Simplification of sign bit logic function

| $\hat{a}_{k-1}$ | $\hat{b}_{n-1}$ | $(\hat{a}_{k-1}\hat{b}_{n-1} + \hat{a}_{k-1} + \hat{b}_{n-1})$ | $2^{n+k-2}$ |
|---|---|---|---|
| | | $2^{n+k-1}$ | $2^{n+k-2}$ |
| 0 | 0 | 0 | 0 |
| -1 | 0 | -1 | 1 |
| 0 | -1 | -1 | 1 |
| -1 | -1 | -1 | 1 |

$$Y = \left(\hat{a}_{k-1}2^{k-1} + \sum_{i=0}^{k-2} a_i 2^i\right)\left(\hat{b}_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right) \tag{8}$$

$$Y = \left(\hat{b}_{n-1}2^{n-1}\right)\left(\hat{a}_{k-1}2^{k-1}\right) + \sum_{j=0}^{n-2} b_j 2^j \left(\hat{a}_{k-1}2^{k-1}\right) + \sum_{i=0}^{k-2}\left(\hat{b}_{n-1}2^{n-1}\right)a_i 2^i + \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} b_j a_i 2^{i+j} \tag{9}$$

where $$\sum_{j=0}^{n-2} b_j 2^j \left(\hat{a}_{k-1}2^{k-1}\right) = \hat{a}_{k-1}2^{k+n-2} + a_{k-1}2^{k-1} + \sum_{j=0}^{n-2} \bar{b}_j a_{k-1} 2^{j+k-1} \tag{10}$$

and $$\sum_{i=0}^{k-2}\left(\hat{b}_{n-1}2^{n-1}\right)a_i 2^i = \hat{b}_{n-1}2^{k+n-2} + b_{n-1}2^{n-1} + \sum_{i=0}^{k-2} b_{n-1}\bar{a}_i 2^{i+n-1} \tag{11}$$

then
$$Y = \left(\widehat{a}_{k-1}\widehat{b}_{n-1} + \widehat{a}_{k-1} + \widehat{b}_{n-1}\right) 2^{k+n-2} + a_{k-1}2^{k-1} + b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} \overline{b}_j a_{k-1} 2^{j+k-1} + \sum_{i=0}^{k-2} b_{n-1}\overline{a}_i 2^{i+n-1} + \sum_{j=0}^{n-2}\sum_{i=0}^{k-2} b_j a_i 2^{i+j} \quad (12)$$

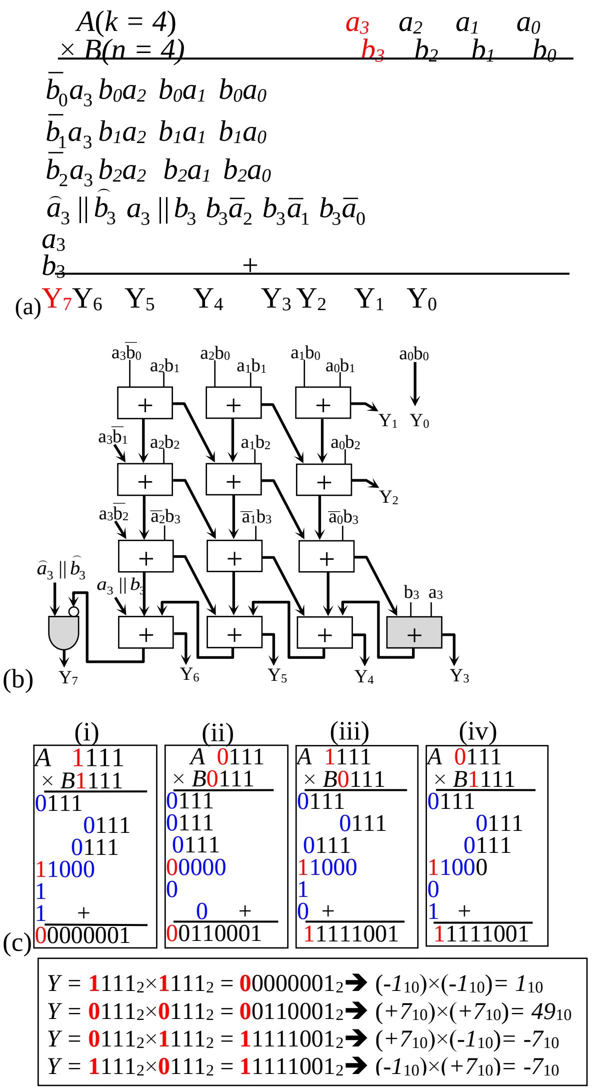
Proposed SN×SN:
$$Y = \underbrace{\left(\widehat{a}_{k-1} \,||\, \widehat{b}_{n-1}\right) 2^{k+n-1}}_{Sign\ bit} + \underbrace{\left(a_{k-1} \,||\, b_{n-1}\right) 2^{k+n-2}}_{MSB} + \underbrace{\left(\sum_{j=0}^{n-2} a_{k-1}\overline{b}_j 2^{j+k-1}\right) + a_{k-1}2^{k-1}}_{Multiplication:\ a_{k-1}\ and\ two's\ complement\ of\ B} + \underbrace{\left(\sum_{i=0}^{k-2} \overline{a}_i b_{n-1} 2^{i+n-1}\right) + b_{n-1}2^{n-1}}_{Multiplication:\ b_{n-1}\ and\ two's\ complement\ of\ A} + \underbrace{\sum_{j=0}^{n-2}\sum_{i=0}^{k-2} a_i b_j 2^{i+j}}_{Multiplication:\ b_j a_i} \quad (13)$$

Table 3.
Multiplication algorithm

```
Algo-1.(SN×SN) Multiplication Algorithm:
(+B)×(+A); (+B)×(-A); (-B)×(+A) and (-B)×(-A)
Input: signed A(k bits), signed B(nbits)
Output: signed Y(p bits)
Process :
1       $Y(p-1:0) \leftarrow 0$;
2       $S_y \leftarrow A(k-1) || B(n-1)$;
3       $Y(n-1) \leftarrow S_y$;
4     if $A(k-1) = 1$
5$Y \leftarrow Y + \{Comp(B(n-2:0)) + A(k-1)\}$;
6       endif
7       if $B(n-1) = 1$
8        $Y \leftarrow Y + \{Comp(A(k-2:0)) + B(n-1)\}$;
9       endif
10      for $j = (n-2)$ downto 0
11        $Y \leftarrow Y \ll 1$;
12      if $B(j) = 1$
13      $Y \leftarrow Y + A(k-2:0)$;
14      endif
15      endforj
16      $Y(p-1) \leftarrow (S_y \,\&\, \overline{Y(p-1)})$;
endprocess
```

## B. Multiplication Structure and Algorithm

The equation (13) multiplication process can be implemented in the software mode using the algorithm *Algo-1* (Table 3). Both variables *A* (*k* bits) and *B* (*n* bits) are signed binary numbers, each having one sign bit and *k*-1, and *n*-1 magnitude bits. The shift-left and AND logic operations are respectively symbolized by "«" and "&". This algorithm consists of five parts. First, steps 1–3 are accumulator initialization and determining MSB value: $S_y = \left(a_{k-1} \,||\, b_{n-1}\right)$, and then save it to the accumulator at position *Y*(*n*-1). The second and third parts in steps 4–6 and steps 7–9 are the multiplication processes of $\widehat{a}_{k-1}\, 2^{k-1}$×(two's complement of *B*) and $\widehat{b}_{n-1}\, 2^{n-1}$×(two's complement of *A*), respectively. Each of these processes is carried out when the conditions are met, and their results are added to the accumulator *Y*.Fourth, steps 10–15 are the multiplication process of $b_j 2^j \times A(k-2:0)$. The last part (step 16) sets up the sign bit value of the multiplication result $(\widehat{a}_{K-1} \,||\, \widehat{b}_{N-1})2^{N+K-1} + C_{out}$, and is done by the logic process $S_y \& \overline{Y(p-1)}$. It means if carry-out ($C_{out}$)at*Y*(*p*-1) = "1" and $S_y$ = "1" (borrow), then the sign bit at *Y*(*p*-1)is set to be "0", otherwise if carry-out at *Y*(*p*-1) = "0" then the sign bit at *Y*(*p*-1)= $S_y$. Finally, the multiplication result consists of *Y*(*p*-1)as the sign bit and *Y*(*p*-2:0) as the magnitude bits.



$Y$ = 1111₂×1111₂ = 00000001₂➔ (-1₁₀)×(-1₁₀)= 1₁₀

$Y$ = 0111₂×0111₂ = 00110001₂➔ (+7₁₀)×(+7₁₀)= 49₁₀

$Y$ = 0111₂×1111₂ = 11111001₂➔ (+7₁₀)×(-1₁₀)= -7₁₀

$Y$ = 1111₂×0111₂ = 11111001₂➔ (-1₁₀)×(+7₁₀)= -7₁₀

Figure 5. SN×SN structure: (a) shift-and-add, (b) array structure, (c) examples of SN×SN

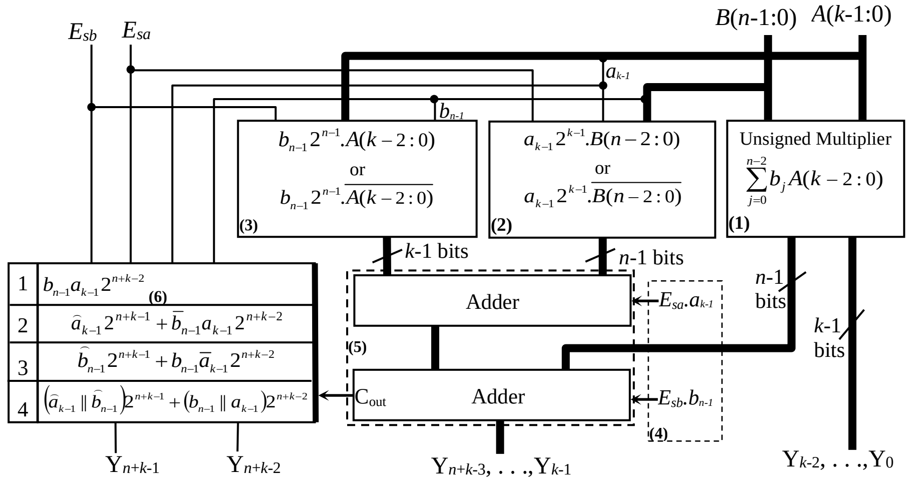
The Algo-1 conforms to the serial-parallel or shift-and-add structure in Figure 5a. For $n\times k$ bits multiplier, the hardware implementation of this structure requires $n$ clock cycles. Another basic implementation that just needs one clock cycle is a parallel or array multiplier with ripple carry adder (RCA), as given in Figure 5b. Four examples of its binary multiplication process (for $k=n=4$) and decimal numbers are given in Figure 5c. The bits in blue represent the two's complement value, and the red ones indicate the sign bit. One can see that shift-and-add and array multiplier structures in Figure 5a and Figure 5b are simpler than Baugh-Wooley’s multiplier in Figure 3.This also makes its hardware easier to implement into the FPGA form of a serial-parallel multiplier and a parallel multiplier.

## C. Proposed Universal Multiplier Design

So that all the multiplication processes in Table 1 can be carried out, based on the proposed signed multiplier, a universal multiplier is developed by including two enable bits of $E_{sa}$ and $E_{sb}$. These two bits control the multiplication function corresponds to the data types of both operands A and B. Bit $E_{sa}$ means Enable sign/unsigned numbers of A: if $E_{sa}$ = "0", then A is unsigned numbers, and if $E_{sa}$ = "1" then A is signed numbers. The same goes for the $E_{sb}$ bit as Enable sign/unsigned numbers of B.

The general schematic design of the proposed universal multiplier is shown in Figure 6. This scheme consists of six main blocks. Block 1 is an unsigned multiplier of B(n-2:0) × A(k-2:0). Blocks 2 and 4 control the multiplication bits of $a_{k-1}2^{k-1}\times B(n\text{-}2{:}0)$ when $E_{sa}$ = "0" or $a_{k-1}2^{k-1}\times \overline{B(n-2:0)}+a_{k-1}$ (2's complement of B) when $E_{sa}$ = "1". Blocks 3 and 4 set the multiplication of $b_{n-1}2^{n-1}\times A(k\text{-}2{:}0)$ when $E_{sb}$ = "0" or $b_{n-1}2^{n-1}\times \overline{A(k-2:0)}+b_{n-1}$ (2’s complement of $A$) when $E_{sb}$ = "1". The last block 6 generates the sign and MSB bits and controls their addition to the carry-out from block 5.Finally, the product Y = B×A has $n+k$-1 bits consisting of $Y_{n+k-1}$ and $Y_{n+k-2}$, . . ., $Y_0$. $Y_{n+k-1}$ acts as the MSB of magnitude bits when $E_{sa}$ = $E_{sb}$ = "0" and will become the signed bit when $E_{sa}$ = "1" and/or $E_{sb}$ = "1".

The control function of blocks 2, 3, 4 and 6 can be explained as shown in table 4. In the first row, $E_{sa}$ and $E_{sb}$ bits are set to "0" if both A and B are unsigned numbers. In this case, blocks 2-4 and 3-4 process partial multiplication bits of $a_{k-1}2^{k-1}.B(n\text{-}2{:}0)+0$ and $b_{n-1}2^{n-1}.A(k\text{-}2{:}0)$ + 0, respectively. Block 6 generates MSB = $a_{k-1}.b_{n-1}$ without sign bit. Furthermore, if $E_{sa}$ and $E_{sb}$ bits are set to "1" and "0", then A and B are considered as signed and unsigned numbers, respectively. In this state, blocks 2-4 deliver $a_{k-1}2^{k-1}.\ \overline{B(n-2:0)}+a_{k-1}$ (two’s complement of B), blocks 3-4 result $b_{n-1}2^{n-1}.A(k\text{-}2{:}0)$ + 0, and block 6 yield MSB = $a_{k-1}.\overline{b_{n-1}}$ and SB = $\hat{a}_{k-1}$. In the third row, $E_{sa}$ and $E_{sb}$ bits are set to "0" and "1", then A and B are expressed as unsigned and signednumbers, respectively. So that,blocks 2-4 and blocks 3-4generate $a_{k-1}2^{k-1}.B(n\text{-}2{:}0)$ + 0 and $b_{n-1}2^{n-1}.\ \overline{A(k-2:0)}+b_{n-1}$ (two’s complement of A), and block 6 evokes MSB = $\overline{a_{k-1}}.b_{n-1}$ and SB = $\hat{b}_{n-1}$.The last line, whenboth A and B are signed numbers, then $E_{sa}$ and $E_{sb}$ bitsmust be set to "1". For this condition, blocks 2-3 and 2-4process two’s complement of A and B, and block 6 produces MSB = $\hat{a}_{k-1}\,\|\,\hat{b}_{n-1}$ and SB = $a_{k-1}\|b_{n-1}$.



$E_{sb}$ $E_{sa}$
$B(n\text{-}1{:}0)$ $A(k\text{-}1{:}0)$
$a_{k-1}$
$b_{n-1}$
$b_{n-1}2^{n-1}.A(k-2:0)$ or $b_{n-1}2^{n-1}.\overline{A(k-2:0)}$ (3)
$a_{k-1}2^{k-1}.B(n-2:0)$ or $a_{k-1}2^{k-1}.\overline{B(n-2:0)}$ (2)
Unsigned Multiplier $\sum_{j=0}^{n-2} b_j A(k-2:0)$ (1)
$k$-1 bits
$n$-1 bits
$n$-1 bits
$k$-1 bits
1 $b_{n-1}a_{k-1}2^{n+k-2}$ (6)
2 $\hat{a}_{k-1}2^{n+k-1}+\overline{b}_{n-1}a_{k-1}2^{n+k-2}$
3 $\hat{b}_{n-1}2^{n+k-1}+b_{n-1}\overline{a}_{k-1}2^{n+k-2}$
4 $(\hat{a}_{k-1}\|\hat{b}_{n-1})2^{n+k-1}+(b_{n-1}\|a_{k-1})2^{n+k-2}$
Adder
$E_{sa}.a_{k-1}$
(5)
$C_{out}$
Adder
$E_{sb}.b_{n-1}$
(4)
$Y_{n+k-1}$
$Y_{n+k-2}$
$Y_{n+k-3}, \ldots, Y_{k-1}$
$Y_{k-2}, \ldots, Y_0$

Figure 6. General diagram of the proposed universal multiplier

Table 4.
Enabling bits for multiplication control functions

| Model | Bits Control | | Block 2 | Block 3 | Block 4 (LSB) | | Block 6 | | Multiplication Functions |
|---|---|---|---|---|---|---|---|---|---|
| | $E_{sa}$ | $E_{sb}$ | | | $2^{k-1}$ | $2^{n-1}$ | *MSB* | *Sign Bit* | |
| 1 | 0 | 0 | $a_{k-1}2^{k-1}B(n-2{:}0)$ | $b_{n-1}2^{n-1}A(k-2{:}0)$ | 0 | 0 | $a_{k-1}.b_{n-1}$ | - | *USN×USN* |
| 2 | 1 | 0 | $a_{k-1}2^{k-1}.\overline{B(n-2:0)}$ | $b_{n-1}2^{n-1}A(k-2{:}0)$ | $a_{k-1}$ | 0 | $a_{k-1}.\overline{b_{n-1}}$ | $\hat{a}_{k-1}$ | *USN×SNN* and *USN×SPN* |
| 3 | 0 | 1 | $a_{k-1}2^{k-1}B(n-2{:}0)$ | $b_{n-1}2^{n-1}.\overline{A(k-2:0)}$ | 0 | $b_{n-1}$ | $\overline{a_{k-1}}.b_{n-1}$ | $\hat{b}_{n-1}$ | *SNN×USN* and *SPN×USN* |
| 4 | 1 | 1 | $a_{k-1}2^{k-1}.\overline{B(n-2:0)}$ | $b_{n-1}2^{n-1}.\overline{A(k-2:0)}$ | $a_{k-1}$ | $b_{n-1}$ | $a_{k-1} \| b_{n-1}$ | $\hat{a}_{k-1}\|\| \hat{b}_{n-1}$ | *SPN×SPN, SPN×SNN, SNN×SPN,* and *SNN×SNN* |

(a)

| $a_{n-1}b_{n-1}$ / $E_{sa}E_{sb}$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | |
| 01 | | 1 | | |
| 11 | | 1 | 1 | 1 |
| 10 | | | | 1 |

(b)

| $a_{n-1}b_{n-1}$ / $E_{sa}E_{sb}$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | 1 | 1 | |
| 11 | | 1 | 1 | 1 |
| 10 | | | 1 | 1 |

Figure 7. Karnaugh maps of MSBC and SBC

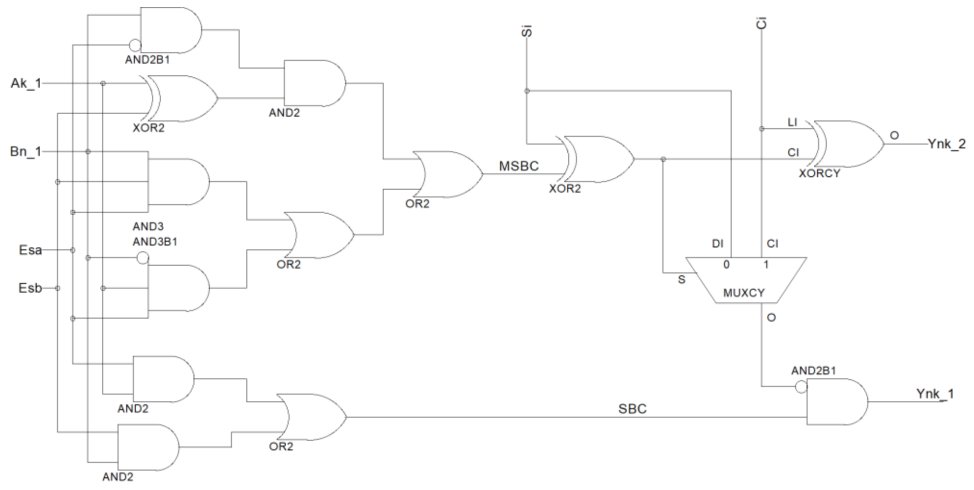


Figure 8. Digital circuit schematic of Block 6

Logically based on table 4, columns 2, 3, 8, and 9, the control function of the MSB (MSBC) and the sign bit (SBC) can be formulated using the Karnaugh map as given in Figure 7. Then from each of these Karnaugh maps, equations 16 and 17 are obtained. These equations can be transformed into a digital circuit schematic, as shown in Figure 8. This schematic is accompanied by one full adder (FA), one

$$MSBC = \overline{E_{sa}}.b_{n-1}(E_{sb} \oplus a_{k-1}) \,||\, (E_{sa}.E_{sb}.b_{n-1}) \,||\, (E_{sa}.a_{k-1}.\overline{b_{n-1}}) \quad , \tag{16}$$

$$SBC = (E_{sa}.a_{k-1}) \,||\, (E_{sb}.b_{n-1}) \quad , \tag{17}$$

gate to complete the two last bits of MSB $=Y_{n+k-2}$, and sign bit SB = $Y_{n+k-1}$. The FA consists of one MUXCY and one XORCY component. $S_i$ and $C_i$ bits are the two last bits of block 5 in Figure 6. Furthermore, the digital circuit schematics of blocks 2 and 3 are shown in Figures 9 and 10, respectively. The XOR gates act as one's complement when $E_{sa}$= 1 (Figure 9) and $E_{sb}$= 1 (Figure 10). Otherwise, they work like buffer gates.

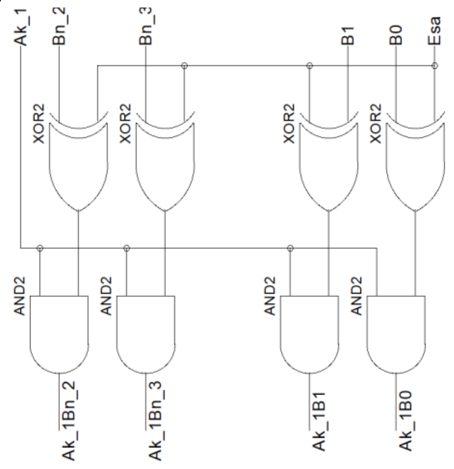


Figure 9. Digital circuit schematic of Block 2

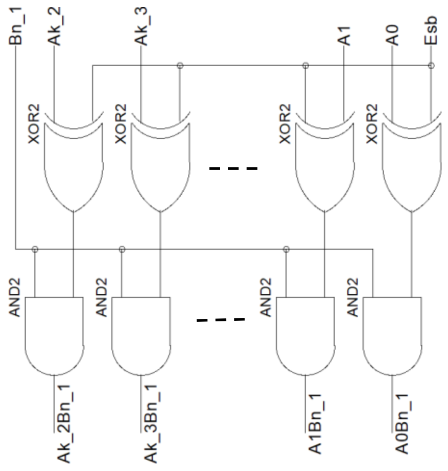


Figure 10. Digital circuit schematic of Block 3

## D. Implementation and Simulation Results

SN×SN and universal multipliers have been implemented in FPGA using two basic serial-parallel and similar methods. The preliminary implementation is done using Xilinx ISE DesignSuite 14.7 through the schematic entry of logic diagrams and based on logic elements of LUTs (LUT6 and LUT5), carry chain Carry4, fast carry logics: MUXCYs and XORCYs, as explained in [29], [30], [31], [32]. All design results were synthesized for the Kintex-7 XC7K70T-FBG676 (-3 speed grade) and post-place-and-route with the strategy set to "ISE XST Synthesis Defaults", and they were not constrained. Figure 11 shows the FPGA RTL schematic of 8x8 bits serial-parallel universal multiplier (SPUMult8x8) that has inputs: multiplicand A (7:0), multiplier B (7: 0), control bits Esa and Esb, Rst (reset), Clk (clock), and LoaD (load data A and B), and output: sixteen bits Y(15: 0) of multiplication result.ThisSPUMult8x8 circuit consists of six main boxes. The first two boxes on the top-left are the counter control and shift-register for multiplier data B. The second three boxes on the top-right are the 2×8 bits shift-register to store the sum of the partial multiplication for each clock cycle and the MSBC and SBC bits controller. The next box at the bottom-middle is the partial multiplier and adders (shift-and-add), where its content is shown on the right of Figure 11.

The simulation results of this SPUMult8x8 multiplier are carried out by applyingthe binary numbers of A = $11111111_2$ and B = $11111111_2$ for all combinations of bits control $E_{sa}$ and $E_{sb}$. Figures 12a to 12dshow its simulation results. The multiplication process starts after data loading, asserted by signals LoaD = "1" and clock rising edge. At each clock, the product value continues to change, until the end process at the eighth clock.In Figure 12a, both $E_{sa}$ and $E_{sb}$ are set to "0", which means A and B are considered as unsigned numbers: A = $11111111_2$= $255_{10}$ and B= $11111111_2$= $255_{10}$.The data A and B are displayed in radix 2 and product Y is in radix 10.So, the product of Y = B×A = $(255\times255)_{10}$ = $65025_{10}$.Next in Figure 12b, for signals $E_{sa}$ = "0" and $E_{sb}$ = "1", then A and B reflectthe unsigned and signed numbers: A = $11111111_2$ = $255_{10}$, B = $11111111_2$ = $-1_{10}$ and the product of Y = $-255_{10}$. Furthermore, when $E_{sa}$ = "1" and $E_{sb}$ = "0", then A and B arethe signed and unsigned numbers: A =$11111111_2$ = $-1_{10}$, B =$11111111_2$ = $255_{10}$, and the product of Y = -$255_{10}$. Finally, if both $E_{sa}$ and $E_{sb}$ are set to "1", then A =$11111111_2$ = $-1_{10}$, B =$11111111_2$ = $-1_{10}$, and the product of Y = $-1_{10}$. Other example results are summarized in table 4.

Figure 13 shows the FPGA RTL schematic of 8x8 bits parallel universal multiplier (UMult8x8) that has inputs: multiplicand A(7:0), multiplier B(7:0), control bits Esa and Esb, and output: sixteen bits Y(15: 0) of multiplication result. This multiplier consists of six blocks, according to Figure 6. The block in the middle corresponds to block 1 of Fig. 6. The blocks on the top-left and bottom-left represent blocks (2-4) and (3-4), respectively. The blocks in the middle-top and middle-bottom are blocks (5) and blocks (6). Its simulation results are given in Figure 14. Each example is described in Table 4.Example No. 1 for the control of the bit are set to $E_{sa}$ = “0” and $E_{sb}$ = “0”, this UMult8x8 circuit only works as UNS×UNS multiplier. So on until the example in No. 4, when both $E_{sa}$ and $E_{sb}$ are set to "1", this circuit can perform four multiplication functions: SNN×SNN, SNN×SPN, SPN×SNN, and SPN×SPN.

Referring to the results of hardware implementation and simulation of both PSUMult8x8 and UMult8x8 circuits, it can be concluded that the proposed universal multiplier can execute all the multiplication functions described in Table 1.

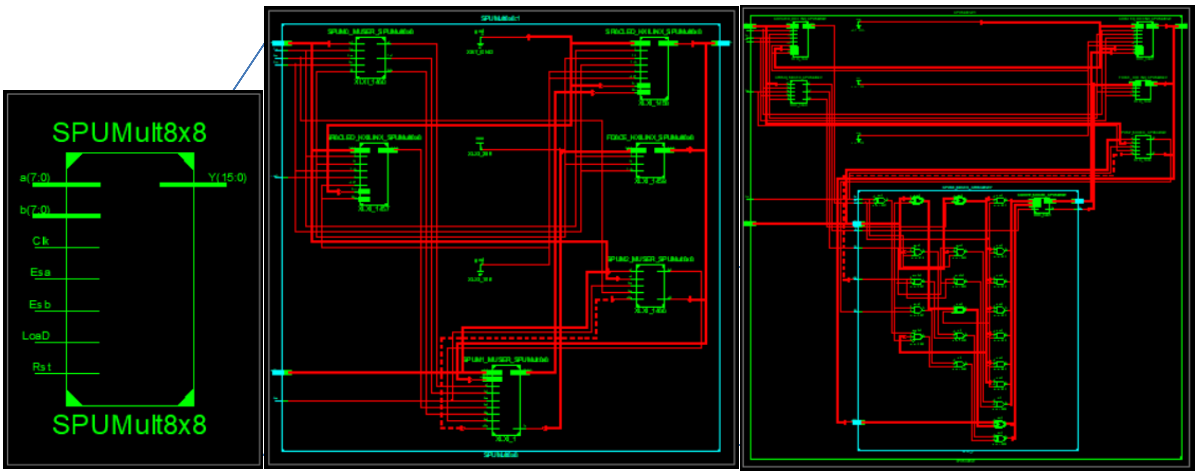
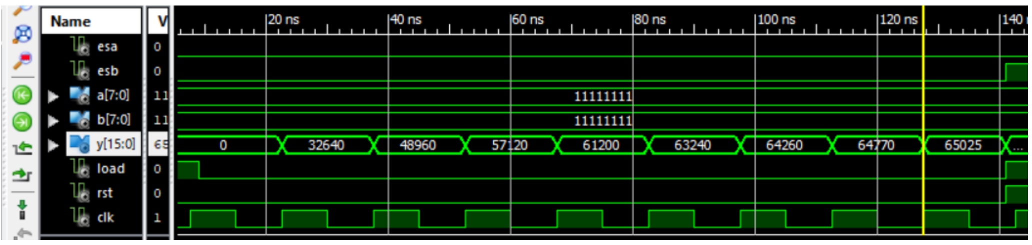
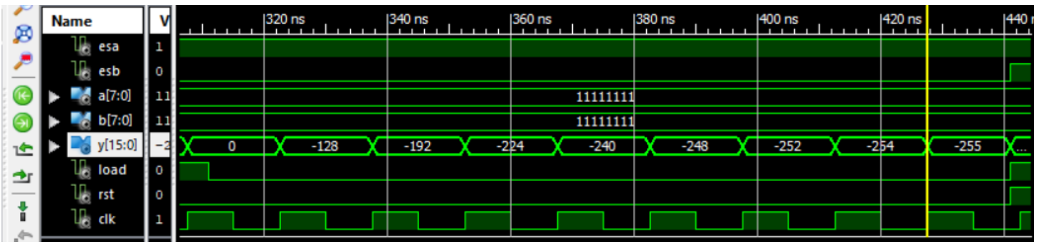


Figure 11. RTL schematic of 8x8 bits serial-parallel universal multiplier

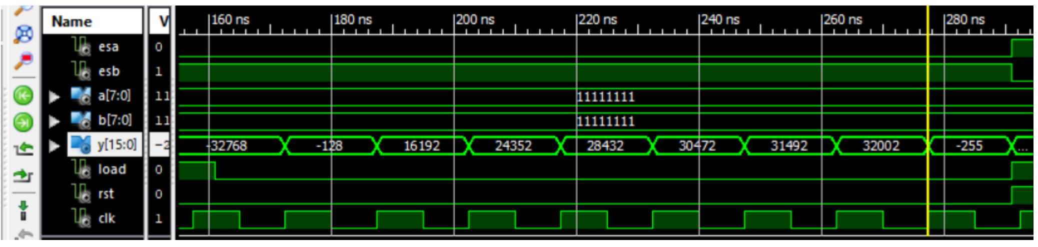
(a)



(b)



(c)



(d)

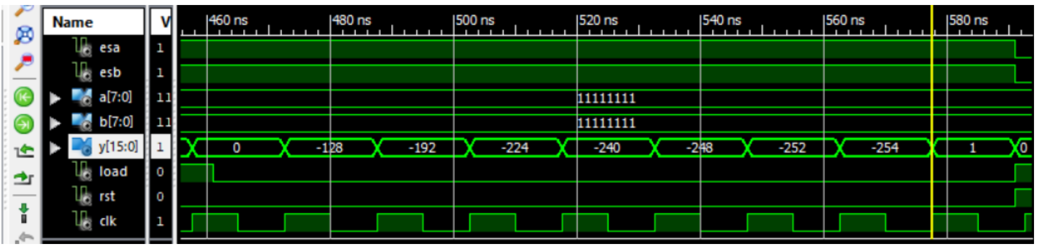


Figure 12. Simulation results of 8x8 bits serial-parallel universal multiplier: (a). $E_{sa}$="0", $E_{sb}$="0", (b). $E_{sa}$="0", $E_{sb}$="1", (c). $E_{sa}$= "1", $E_{sb}$ = "0", (d). $E_{sa}$="1", $E_{sb}$="1"

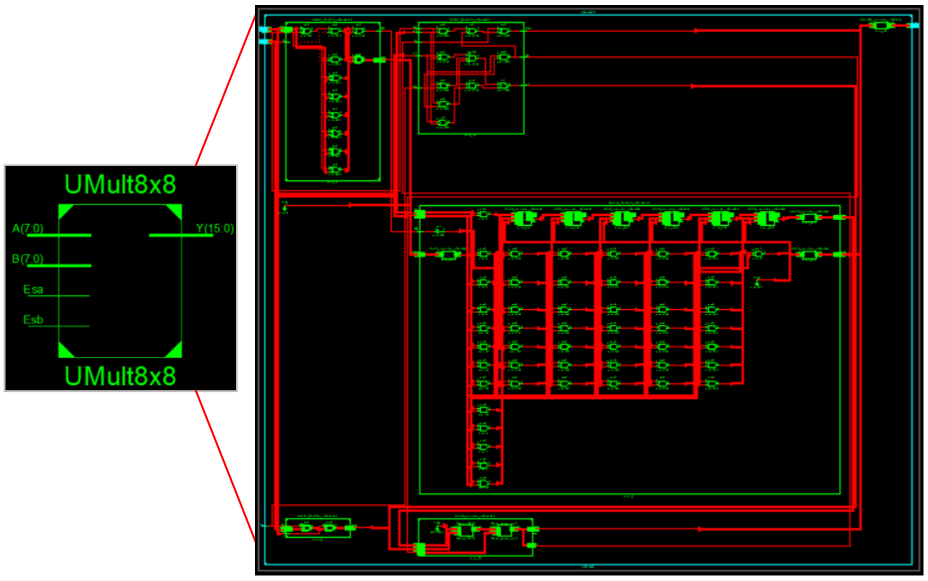


Figure 13. RTL schematic of 8x8 bits parallel universal multiplier

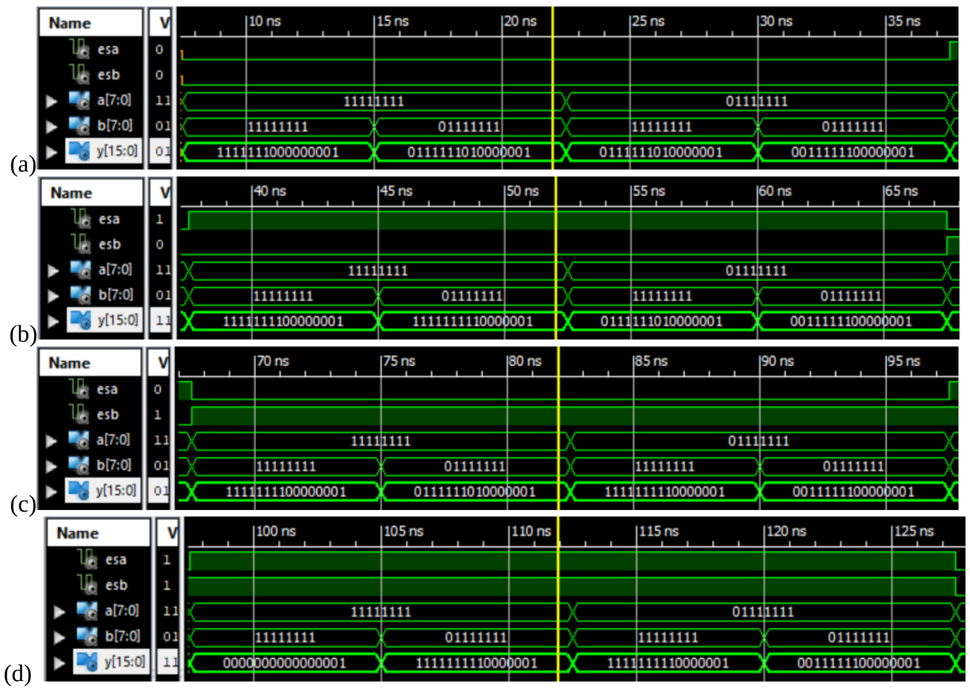


Figure 14. Simulation results of 8x8 bits parallel universal multiplier: (a). $E_{sa}$="0", $E_{sb}$="0", (b). $E_{sa}$="0", $E_{sb}$="1", (c). $E_{sa}$= "1", $E_{sb}$ = "0", (d). $E_{sa}$="1", $E_{sb}$="1"

Table 5.
Simulation results of proposed serial-parallel and parallel universal multiplier

| No. | $E_{sa}E_{sb}$ | Multiplication types | A (8 bits) | B (8 bits) | Y = B×A (16 bits) |
|---|---|---|---|---|---|
| 1 | 0 0<br>Figure 12a<br>Figure 14a | USN × USN | $11111111_2 = 255_{10}$ | $11111111_2 = 255_{10}$ | $1111111000000001_2 = 65025_{10}$ |
| | | | $11111111_2 = 255_{10}$ | $01111111_2 = 127_{10}$ | $0111111010000001_2 = 32385_{10}$ |
| | | | $01111111_2 = 127_{10}$ | $11111111_2 = 255_{10}$ | $0111111010000001_2 = 32385_{10}$ |
| | | | $01111111_2 = 127_{10}$ | $01111111_2 = 127_{10}$ | $0011111100000001_2 = 16129_{10}$ |
| 2 | 1 0<br>Figure 12b<br>Figure 14b | USN × SNN | $11111111_2 = -1_{10}$ | $11111111_2 = 255_{10}$ | $1111111100000001_2 = -255_{10}$ |
| | | | $11111111_2 = -1_{10}$ | $01111111_2 = 127_{10}$ | $1111111110000001_2 = -127_{10}$ |
| | | USN × SPN | $01111111_2 = +127_{10}$ | $11111111_2 = 255_{10}$ | $0111111010000001_2 = 32385_{10}$ |
| | | | $01111111_2 = +127_{10}$ | $01111111_2 = 127_{10}$ | $0011111100000001_2 = 16129_{10}$ |
| 3 | 0 1<br>Figure 12c<br>Figure 14c | SNN × USN | $11111111_2 = 255_{10}$ | $11111111_2 = -1_{10}$ | $1111111100000001_2 = -255_{10}$ |
| | | SPN × USN | $11111111_2 = 255_{10}$ | $01111111_2 = +127_{10}$ | $0111111010000001_2 = 32385_{10}$ |
| | | SNN × USN | $01111111_2 = 127_{10}$ | $11111111_2 = -1_{10}$ | $1111111110000001_2 = -127_{10}$ |
| | | SPN × USN | $01111111_2 = 127_{10}$ | $01111111_2 = +127_{10}$ | $0011111100000001_2 = 16129_{10}$ |
| 4 | 1 1<br>Figure 12d | SNN × SNN | $11111111_2 = -1_{10}$ | $11111111_2 = -1_{10}$ | $0000000000000001_2 = 1_{10}$ |
| | | SPN × SNN | $11111111_2 = -1_{10}$ | $01111111_2 = +127_{10}$ | $1111111110000001_2 = -127_{10}$ |

| | | | | |
|---|---|---|---|---|
| Figure 14d | SNN × SPN | $01111111_2 = +127_{10}$ | $11111111_2 = -1_{10}$ | $1111111110000001_2 = -127_{10}$ |
| | SPN × SPN | $01111111_2 = +127_{10}$ | $01111111_2 = +127_{10}$ | $0011111100000001_2 = 16129_{10}$ |

The FPGA resources used for PSUMult8x8 and UMult8x8 multipliers are given in table 6. The PSUMult8x8 multiplier needs 2 Carry4, one Counter 4 bits, 24 Flip-flops, and occupied 21 LUTs. Its logic and route delays are 0.831 ns and 2.087 ns, respectively, and entail 8-hour cycles to deliver its product. Whereas the UMult8x8 multiplier requires one cycle to convey its result but involves more resources, 60 LUTs including 14 Carry4, with logic and route time delays are 3.677 ns and 5.189 ns.

Table 6.
Occupied FPGA resources for the proposed SN×SN and universal multipliers

| Implementation Technique | Multiplier function | *n*×*k* bits Multiplier | Carry logic (Carry4) | Counter | Occupied Flip-flops | Occupied LUTs | Logic Delay (ns) | Route Delay (ns) | Clock cycle |
|---|---|---|---|---|---|---|---|---|---|
| Serial-Parallel | SN×SN | 8×8 | 2 | 1 (4 bits) | 28 | 19 | 0.769 | 1.971 | 8 |
| Parallel | SN×SN | 8×8 | 14 | - | - | 58 | 3.600 | 4.959 | 1 |
| Serial-Parallel | Universal | 8×8 | 2 | 1 (4 bits) | 28 | 21 | 0.831 | 2.087 | 8 |
| Parallel | Universal | 8×8 | 14 | - | - | 60 | 3.677 | 5.189 | 1 |

## IV. Conclusion

Three novelties have been proposed in this paper. First is the new mathematical equation and algorithm of signed numbers multiplication, which can execute four multiplication types: SNN×SNN, SNN×SPN, SPN×SNN, and SPN×SPN. The second is an architectural design of a universal multiplier that is developed based on the proposed equation. The last is the two bits ($E_{sa}$ and $E_{sb}$) control system design, which adjusts the two's complement process of A, two's complement of B, the MSB bit, and the signed or unsigned bit. This bits control system enables the proposed universal multiplier to perform all binary multiplications: USN×USN, USN×SNN, USN×SPN, SNN×USN, SPN×USN, SNN×SNN, SNN×SPN, SPN×SNN, and SPN×SPN.

The universal multiplier was implemented using two classic serial-parallel and similar techniques into the FPGA. The 8x8 bits serial-parallel multiplier occupies 20 LUTs and needs 8-hour cycles to deliver its product with a logic delay of 0.831 ns. At the same time, the 8x8 bits parallel multiplier requires 60 LUTs and involves one clock cycle with a logic delay of 3.677 ns.

## Acknowledgment

We would like to thank Gunadarma University and DP2M of the Directorate General of Higher Education, Minister for Education and Culture of the Republic of Indonesia. They have provided support in conducting research and publications.

## References

[1] CHEN, Y., DUFFNER, S., STOIAN, A., DUFOUR, J.-Y., and BASKURTA, A. (2018) Deep and low-level feature-based attribute learning for person re-identification. *Image and Vision Computing, Elsevier*, 79, pp. 25–34.

[2] CHENG, X., LU, J., FENG, J., YUAN, B., and ZHOU, J. (2018) Scene recognition with objectness. *Pattern Recognition*, 74, pp. 474–487.

[3] ZHANG, J., SHAO, K., and LUO, X. (2018) Small sample image recognition using improved Convolutional Neural Network. *Journal of Visual Communication and Image Representation*, 55, pp. 640–647.

[4] SARIKAN, S.S., OZBAYOGLU, A.M., and ZILCIA, O. (2017) Automated vehicle classification with image processing and computational intelligence. *Procedia Computer Science*, 114, pp. 515–522.

[5] QAYYUM, A., ANWAR, S.M., AWAIS, M., and MAJID, M. (2017) Medical image retrieval using deep convolutional neural network. *Neurocomputing*, 266, pp. 8–20.

[6] GONG, L., WANG, C., LI, X., CHEN, H., and ZHOU, X. (2018) MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. *Institute of Electrical and Electronics Engineers Transactions.*

*Computer Aided Design Integrated Circuits and System*, 37(11), pp. 2601–2612.

[7] CHERVYAKOV, N.I., LYAKHOV, P.A., and VALUEVA, M.V. (2017) Increasing of Convolutional Neural Network performance using residue number system. In *International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, pp. 135–140.

[8] SHAWAHNA, A., SAIT, S.M., and EL-MALEH, A. (2019) FPGA-based accelerators of deep learning networks for learning and classification: A review. *Institute of Electrical and Electronics Engineers Access*, 7, pp. 7823–7859.

[9] SIM, H. and LEE, J. (2017) A New Stochastic Computing Multiplier with Application to Deep Convolutional Neural Networks. In *54th Design Automation Conference*, pp. 1-6.

[10] RENTERIA-CEDANO, J., RIVERA, J., SANDOVAL-IBARRA, F., ORTEGA-CISNEROS, S., and LOO-YAU, R. (2019) SoC Design Based on a FPGA for a Configurable Neural Network Trained by Means of an EKF. *Electronics*, 8 (761), pp 1-19.

[11] XILINX (2020) FPGA Acceleration of Matrix Multiplication for Neural Networks. Available from https://www.xilinx.com/support/documentation/application_notes/xapp1332-neural-networks.pdf

[12] WILSON, P. (2015) Multiplication. In *Design Recipes for FPGAs (Second Edition)*. Amsterdam: Elsevier, p. 345-353.

[13] RAJMOHAN, V. and UMA MAHESWARI, O. (2016) Design of Compact Baugh-Wooley Multiplier Using Reversible Logic. *Circuits and Systems*, 7 (8), pp. 1522-1529.

[14] KEOTE, R.S. and KARULE, P.T. (2018) Performance Analysis of Fixed Width Multiplier using Baugh Wooley Algorithm. *International Organization of Scientific Research Journal of Very Large-Scale Integration and Signal Processing*, 8 (3), pp. 31-38.

[15] ULLAH, S., NGUYEN, T.D.A., and KUMAR, A. (2021) Energy-efficient low-latency signed multiplier for FPGA-based hardware accelerators. *Institute of Electrical and Electronics Engineers - Embedded Systems Letters*, 13 (2), pp. 41-44.

[16] OUDJIDA, A.K., CHAILLET, N., and BERRANDJIA, M.L. (2015) RADIX-2r arithmetic for multiplication by a constant: further results and improvements. *Institute of Electrical and Electronics Engineers Transactions on Circuits and Systems II*, 62 (4), pp. 372–376.

[17] KUMM, M. (2018) Optimal Constant Multiplication Using Integer Linear Programming. *Institute of Electrical and Electronics Engineers Transactions on Circuits and Systems II*, 65 (5), pp. 567-571.

[18] DE DINECHIN, F., FILIP, S-I., KUMM, M., and FORGET, L. (2019) Table-Based versus Shift-And-Add Constant Multipliers for FPGAs. In *26th Symposium on Computer Arithmetic (ARITH)*, pp. 151-158.

[19] KUMM, M., GUSTAFSSON, O., GARRIDO, M., and ZIPF, P. (2018) Optimal Single Constant Multiplication Using Ternary Adders. *Institute of Electrical and Electronics Engineers Transactions on Circuits and Systems II*, 65 (7), pp. 928-932.

[20] MÖLLER, K., KUMM, M., KLEINLEIN, M. and ZIPF, P. (2017) Reconfigurable constant multiplication for FPGAs. Institute of Electrical and Electronics Engineers *Transactions on Computer Aided Design Integrated Circuits and System*, 36 (6), pp. 927-937.

[21] OUDJIDA, A.K., LIACHA, A., BAKIRI, M. and CHAILLET, N. (2016) Multiple Constant Multiplication Algorithm for High-Speed and Low-Power Design. *Institute of Electrical and Electronics Engineers Transactions on Circuits and Systems II*, 63 (2), pp. 176-180.

[22] FENG, F., CHEN, J., and CHANG, C. H. (2015) Hypergraph-based minimum arborescence algorithm for optimizing and optimizing multiple constant multiplications. *Institute of Electrical and Electronics Engineers Transactions on Circuits and Systems I*, 63 (2), pp. 233–244.

[23] LOU, X., YU, Y.J., and MEHER, P.K. (2015) Fine grained critical path analysis and optimization for area-time efficient realization of multiple constant multiplications. *Institute of Electrical and*

*Electronics Engineers Transactions on Circuits and Systems I*, 62 (3), pp. 863–872.

[24] JOSEPH, G.B. and DEVANATHAN, R. (2018) Algorithms for Multiplierless Multiple Constant Multiplication in Online Arithmetic. *Circuits Systems and Signal Processing*, 37, pp. 5127–5142.

[25] HOSEININASAB, S.S. and NIKMEHR H. (2019) Architectures for multiple constant decimal multiplication. *Computers & Electrical Engineering*, 75, pp. 31-45.

[26] KUMM, M. (2016) *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays. 1st edition*. Wiesbaden: Springer Vieweg, pp. 15-81.

[27] LOU, X., YU, Y.J., and MEHER, P.K. (2017) Lower Bound Analysis and perturbation of critical path for area-time efficient multiple constant multiplications. *Institute of Electrical and Electronics Engineers Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(2), pp. 313-324.

[28] LIACHA, A., OUDJIDA, A.K., BAKIRI, M., MONTEIRO, J., and FLORES, P. (2020) Radix-2 recoding with common subexpression elimination for multiple constant multiplication. *Institution of Engineering and Technology: Circuits, Devices & Systems*, 14 (7), pp. 990-994.

[29] WALTERS, E.G. (2016) Array Multipliers for High Throughput in Xilinx FPGAs with 6-Input LUTs. *Multidisciplinary Digital Publishing Institute Computers Journal*, 5 (20), pp. 5-25.

[30] WALTERS, E.G. (2017) Reduced-area constant-coefficient and multiple-constant multipliers for xilinx FPGAs with 6-Input LUTs. *Multidisciplinary Digital Publishing Institute Electronics Journal*, 6 (101), pp. 1-29.

[31] ULLAH, S., SCHMIDL, H., SAHOO, S.S., REHMAN, S., and KUMAR, A. (2021) Area-optimized accurate and approximate softcore signed multiplier architectures. *Institute of Electrical and Electronics Engineers Transactions on Computers*. 70 (3), pp. 384-392.

[32] ULLAH, S., REHMAN, S., SHAFIQUE, M., and KUMAR, A. (2021) High-performance accurate and approximate multipliers for fpga-based hardware accelerators. *Accepted for Institute of Electrical and Electronics Engineers Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1 (1), pp.1-14.

## 参考文:

[1] CHEN, Y., DUFFNER, S., STOIAN, A., DUFOUR, J.-Y., 和 BASKURTA, A. (2018) 用于人员重新识别的深度和低级基于特征的属性学习。图像和视觉计算，爱思唯尔, 79, 第 25–34 页.

[2] CHENG, X., LU, J., FENG, J., YUAN, B., 和 ZHOU, J. (2018) 具有客观性的场景识别。模式识别, 74, 第 474–487 页.

[3] ZHANG, J., SHAO, K., 和 LUO, X. (2018) 使用改进的卷积神经网络进行小样本图像识别。视觉传达与图像表示杂志, 55, 第 640–647 页.

[4] SARIKAN, S.S., OZBAYOGLU, A.M., 和 ZILCIA, O. (2017) 具有图像处理和计算智能的自动车辆分类. 普罗西迪亚计算机科学, 114, 第 515–522 页.

[5] QAYYUM, A., ANWAR, S.M., AWAIS, M., 和 MAJID, M. (2017) 使用深度卷积神经网络的医学图像检索。神经计算, 266, 第 8–20 页.

[6] GONG, L., WANG, C., LI, X., CHEN, H., 和 ZHOU, X. (2018) 面向对象 C 的最小抽象层：一个完全流水线化的现场可编程门阵列加速器，用于卷积神经网络，所有层都映射在芯片上。电气和电子工程师学会会刊. 计算机辅助设计集成电路与系统, 37(11), 第 2601–2612 页.

[7] CHERVYAKOV, N.I., LYAKHOV, P.A., 和 VALUEVA, M.V. (2017) 使用残数系统提高卷积神经网络的性能。在国际工程、计算机和信息科学会议上 (西伯利亚会议), 第 135–140 页.

[8] SHAWAHNA, A., SAIT, S.M., 和 EL-MALEH, A. (2019) 用于学习和分类的基于现场可编程门阵列的深度学习网络加速器：综述。电气和电子工程师协会访问权限, 7, 第 7823–7859 页.

[9] SIM, H. 和 LEE, J. (2017) 应用于深度卷积神经网络的新随机计算乘法器。在第54届设计自动化会议上, 第 1-6 页.

[10] RENTERIA-CEDANO, J., RIVERA, J., SANDOVAL-IBARRA, F., ORTEGA-CISNEROS, S., 和 LOO-YAU, R. (2019) 片上系统设计基于现场可编程门阵列, 用于通过扩展卡尔曼滤波器训练的可配置神经网络。电子产品, 8 (761), 第 1-19 页.

[11] 赛灵思 (2020) 现场可编程门阵列 https://www.xilinx.com/support/documentation/application_notes/xapp1332-neural-networks.pdf

[12] WILSON, P. (2015) 乘法。在现场可编程门阵列的设计配方中。第二版。阿姆斯特丹：爱思唯尔, 第 345-353 页.

[13] RAJMOHAN, V., 和 UMA MAHESWARI, O. (2016) 使用可逆逻辑设计紧凑型鲍-伍利乘法器。电路和系统, 7 (8), 第 1522-1529 页.

[14] KEOTE, R.S. 和 KARULE, P.T. (2018) 使用鲍· 伍利算法的固定宽度乘法器的性能分析。国际科学研究组织超大规模集成与信号处理期刊, 8 (3), 第 31-38 页.

[15] ULLAH, S., NGUYEN, T.D.A., 和 KUMAR, A. (2021) 用于基于现场可编程门阵列的硬件加速器的节能低延迟有符号乘法器。电气和电子工程师协会 - 嵌入式系统快报, 13 (2), 第 41-44 页.

[16] OUDJIDA, A.K., CHAILLET, N., 和 BERRANDJIA, M.L. (2015) 乘以常数的基数-2耳算法：进一步的结果和改进。电气与电子工程师学会电路与系统汇刊 II, 62 (4), 第 372–376 页.

[17] KUMM, M. (2018) 使用整数线性规划的最优常数乘法。电气与电子工程师学会电路与系统汇刊 II, 65 (5), 第 567-571 页.

[18] DE DINECHIN, F., FILIP, S-I., KUMM, M., 和 FORGET, L. (2019) 现场可编程门阵列的基于表与“移位和添加”常数乘法器。第26届计算机算术研讨会, 第 151-158 页.

[19] KUMM, M., GUSTAFSSON, O., GARRIDO, M., 和 ZIPF, P. (2018) 使用三元加法器的最佳单常数乘法。电气与电子工程师学会电路与系统汇刊 II, 65 (7), 第 928-932 页.

[20] MÖLLER, K., KUMM, M., KLEINLEIN, M. 和 ZIPF, P. (2017) 现场可编程门阵列的可重构常数乘法, 电气和电子工程师学会计算机辅助设计集成电路和系统传输交易。计算机辅助设计集成电路与系统, 36 (6), 第 927-937 页.

[21] OUDJIDA, A.K., LIACHA, A., BAKIRI, M. 和 CHAILLET, N. (2016) 用于高速和低功耗设计的多重常数乘法算法。电气与电子工程师学会电路与系统汇刊II, 63 (2), 第 176-180 页.

[22] FENG, F., CHEN, J., 和 CHANG, C. H. (2015) 基于超图的最小树状算法, 用于优化和优化多个常数乘法。电气与电子工程师学会电路与系统汇刊 I, 63 (2), 第 233–244 页.

[23] LOU, X., YU, Y.J., 和 MEHER, P.K. (2015) 细粒度关键路径分析和优化, 以实现多个常数乘法的区域时间高效实现。电气与电子工程师学会电路与系统汇刊 I, 62 (3), 第 863–872 页.

[24] JOSEPH, G.B. 和 DEVANATHAN, R. (2018) 在线算术中无乘数多常数乘法的算法。电路系统和信号处理, 37, 第 5127–5142 页.

[25] HOSEININASAB, S.S. 和 NIKMEHR H. (2019) 多个常数十进制乘法的体系结构。计算机与电气工程, 75, 第 31-45 页.

[26] KUMM, M. (2016) 现场可编程门阵列的多重常数乘法优化。第 1 版。威斯巴登：施普林格视图, 第 15-81 页.

[27] LOU, X., YU, Y.J., 和 MEHER, P.K. (2017) 区域时间高效多重常数乘法的关键路径的下限分析和扰动。电气与电子工程师学会计算机辅助设计集成电路与系统汇刊, 36(2), 第 313-324 页.

[28] LIACHA, A., OUDJIDA, A.K., BAKIRI, M., MONTEIRO, J., 和 FLORES, P. (2020) 带有公共子表达式消除的基数-2 重新编码, 用于多个常数乘法。工程技术学院：电路、器件和系统, 14 (7), 第 990-994 页.

[29] WALTERS, E.G. (2016) 具有 6 输入查找表的赛灵思现场可编程门阵列中用于高吞吐量的阵列乘法器。多学科数字出版研究所计算机期刊, 5 (20), 第 5-25 页.

[30] WALTERS, E.G. (2017) 用于具有 6 输入查找表的赛灵思现场可编程门阵列的缩减面积常数系数和多常数乘法器。多学科数字出版研究所电子期刊, 6 (101), 第 1-29 页.

[31] ULLAH, S., SCHMIDL, H., SAHOO, S.S., REHMAN, S., 和 KUMAR, A. (2021) 面积优化的精确和近似软核有符号乘法器架构。电气和电子工程师学会计算机交易. 70 (3), 第 384-392 页.

[32] ULLAH, S., REHMAN, S., SHAFIQUE, M., 和 KUMAR, A. (2021) 用于基于现场可编程门阵列的硬件加速器的高性能精确和近似乘法器。电子电气工程师学会计算机辅助设计集成电路和系统交易被接受, 1 (1), 第 1-14 页.