Microprocessors and Microsystems 35 (2011) 23–33

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

An iterative logarithmic multiplier

Z. Babić [a], A. Avramović [a], P. Bulić [b,*]

[a] University of Banja Luka, Faculty of Electrical Engineering, Banja Luka, Bosnia and Herzegovina
[b] University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia

ARTICLE INFO

Article history:
Available online 21 July 2010

Keywords:
Computer arithmetic
Digital signal processing
Multiplier
Logarithmic number system
FPGA

ABSTRACT

Digital signal processing algorithms often rely heavily on a large number of multiplications, which is both time and power consuming. However, there are many practical solutions to simplify multiplication, like truncated and logarithmic multipliers. These methods consume less time and power but introduce errors. Nevertheless, they can be used in situations where a shorter time delay is more important than accuracy. In digital signal processing, these conditions are often met, especially in video compression and tracking, where integer arithmetic gives satisfactory results. This paper presents a simple and efficient multiplier with the possibility to achieve an arbitrary accuracy through an iterative procedure, prior to achieving the exact result. The multiplier is based on the same form of number representation as Mitchell's algorithm, but it uses different error correction circuits than those proposed by Mitchell. In such a way, the error correction can be done almost in parallel (actually this is achieved through pipelining) with the basic multiplication. The hardware solution involves adders and shifters, so it is not gate and power consuming. The error summary for operands ranging from 8 bits to 16 bits indicates a very low relative error percentage with two iterations only. For the hardware implementation assessment, the proposed multiplier is implemented on the Spartan 3 FPGA chip. For 16-bit operands, the time delay estimation indicates that a multiplier with two iterations can work with a clock cycle more than 150 MHz, and with the maximum relative error being less than 2%.

1. Introduction

Multiplication has always been a hardware-, time- and power-consuming arithmetic operation, especially for large-value operands. This bottleneck is even more emphasized in digital signal processing (DSP) applications that involve a huge number of multiplications [3,6–8,12–14,18,20,22,25]. In many real-time DSP applications, speed is the prime target and achieving this may be done at the expense of the accuracy of the arithmetic operations. Signal processing deals with signals distorted with the noise caused by non-ideal sensors, quantization processes, amplifiers, etc., as well as algorithms based on certain assumptions, so inaccurate results are inevitable. For example, a frequency leakage causes a false magnitude of the frequency bins in spectrum estimations. The signal-compression techniques incorporate quantization after a cosine or wavelet transform. When transform coefficients are quantized, instead of calculating high-precision coefficients and then truncating them, it is reasonable to spend less resources and produce less accurate results before the quantization. In many signal processing algorithms, which include correlation computations, the exact value of the correlation does not matter; only the maximum of the correlation plays a role. Additional small errors introduced with multipliers, as mentioned in the application described and others, do not affect the results significantly and they can still be acceptable in practice. Other applications that involve a significant number of multiplications are found in cryptography [4,5,10,11,19,26,27]. In applications where the speed of the calculation is more important than the accuracy, truncated or logarithm multiplications seem to be suitable methods [14,21].

1.1. Integer multiplication methods

The simplest integer multiplier computes the product of two n-bits unsigned numbers, one bit at a time. There are n multiplication steps and each step has two parts:

1. If the least-significant bit of the multiplicator is 1, then the multiplicand is added to the product, otherwise zero is added to the product.
2. The multiplicand is shifted left (saving the most significant bit) and the multiplicator is shifted right, discarding the bit that was shifted out.

A detailed implementation and description of this multiplication algorithm are given in [9]. Such an integer multiplication,

* Corresponding author. Tel.: +386 1 4768361; fax: +386 1 4264647.
E-mail address: patricio.bulic@fri.uni-lj.si (P. Bulić).

0141-9331/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2010.07.001

where the least-significant bit of the multiplicator is examined, is known as the radix-2 multiplication.

To speed up the multiplication, we can examine k lower bits of the multiplicand in each step. Usually, the radix-4 multiplication is used, where two least-significant bits of the multiplicand are examined. A detailed explanation of the radix-4 multiplication can be found in [9].

Another way to speed up the integer multiplication is to use many adders. Such an approach typically requires a lot of space on the chip. The well-known implementation of such a multiplier is an array multiplier [9], where $n-2$ n-bits carry-save adders and one n-bits carry-propagate adder are used to implement the n-bits array multiplier.

1.2. Truncated multipliers

Truncated multipliers are extensively used in digital signal processing where the speed of the multiplication and the area- and power-consumptions are important. However, as mentioned before, there are many applications in DSP where high accuracy is not important. The basic idea of these techniques is to discard some of the less significant partial products and to introduce a compensation circuit to reduce the approximation error [13,21,23].

1.3. Logarithmic multiplication methods

Logarithmic multiplication introduces an operand conversion from integer number system into the logarithm number system (LNS). The multiplication of the two operands N_1 and N_2 is performed in three phases, calculating the operand logarithms, the addition of the operand logarithms and the calculation of the antilogarithm, which is equal to the multiple of the two original operands. The main advantage of this method is the substitution of the multiplication with addition, after the conversion of the operands into logarithms. LNS multipliers can be generally divided into two categories, one based on methods that use lookup tables and interpolations, and the other based on Mitchell's algorithm (MA) [17], although there is a lookup-table approach in some of the MA-based methods [16]. Generally, MA-based methods suppressed lookup tables due to hardware-area savings. However, this simple idea has a significant weakness: logarithm and anti-logarithm cannot be calculated exactly, so there is a need to approximate the logarithm and the antilogarithm. The binary representation of the number N can be written as:

$$N = 2^k\left(1+\sum_{i=j}^{k-1}2^{i-k}Z_i\right) = 2^k(1+x) \tag{1}$$

where k is a characteristic number or the place of the most significant bit with the value of '1', Z_i is a bit value at the ith position, x is the fraction or mantissa, and j depends on the number's precision (it is 0 for integer numbers). The logarithm with the basis 2 of N is then:

$$\begin{aligned} log_2(N) &= log_2\left(2^k\left(1+\sum_{i=j}^{k-1}2^{i-k}Z_i\right)\right) = log_2(2^k(1+x)) \\ &= k + log_2(1+x) \end{aligned} \tag{2}$$

The expression $log_2(1+x)$ is usually approximated; therefore, logarithmic-based solutions are a trade-off between the time consumption and the accuracy.

This paper presents a simple iterative solution for multiplication with the possibility to achieve an arbitrary accuracy through an iterative procedure, based on the same form of numbers representation as Mitchell's algorithm. The proposed multiplication algorithm uses different error correction formulas than MA. In such a way, the error correction can be started with a very small delay after the main computation and can run almost in parallel with the main computation. This is achieved through pipelining.

The paper is organized as follows: Section 2 presents the basic Mitchell's algorithm and its modifications, with their advantages and weaknesses. Section 3 describes the proposed solution. In Section 4 the hardware implementations of the proposed algorithm are discussed. Section 5 gives a detailed error analysis and the experimental evaluation of the proposed solution. Section 6 shows the usability of the proposed multiplier and Section 7 draws a conclusion.

2. Mitchell's algorithm based multipliers

A logarithmic number system is introduced to simplify multiplication, especially in cases when the accuracy requirements are not rigorous. In LNS two operands are multiplied by finding their logarithms, adding them, and after that looking for the antilogarithm of the sum.

One of the most significant multiplication methods in LNS is Mitchell's algorithm [17]. An approximation of the logarithm and the antilogarithm is essential, and it is derived from a binary representation of the numbers (1).

The logarithm of the product is

$$\log_2(N_1\cdot N_2) = k_1 + k_2 + \log_2(1+x_1) + \log_2(1+x_2) \tag{3}$$

The expression $\log_2(1+x)$ is approximated with x and the logarithm of the two numbers' product is expressed as the sum of their characteristic numbers and mantissas:

$$\log_2(N_1\cdot N_2) \approx k_1 + k_2 + x_1 + x_2 \tag{4}$$

The characteristic numbers k_1 and k_2 represent the places of the most significant operands' bits with the value of '1'. For 16-bit numbers, the range for characteristic numbers is from 0 to 15. The fractions x_1 and x_2 are in range $[0,1)$.

The final MA approximation for the multiplication (where $P_{true} = N_1\cdot N_2$) depends on the carry bit from the sum of the mantissas and is given by:

$$P_{MA} = (N_1\cdot N_2)_{MA} = \begin{cases} 2^{k_1+k_2}(1+x_1+x_2), & x_1+x_2<1 \\ 2^{k_1+k_2+1}(x_1+x_2), & x_1+x_2 \geqslant 1 \end{cases} \tag{5}$$

The final approximation for the product (5) requires the comparison of the sum of the mantissas with '1'.

The sum of the characteristic numbers determines the most significant bit of the product. The sum of the mantissas is then scaled (shifted left) by $2^{k_1+k_2}$ or by $2^{k_1+k_2+1}$, depending on the x_1+x_2. If $x_1+x_2<1$, the sum of mantissas is added to the most significant bit of product to complete the final result. Otherwise, the product is approximated only with the scaled sum of mantissas. The proposed MA-based multiplication is given in Algorithm 1.

Algorithm 1 (Mitchell's algorithm).

```
1. $N_1$, $N_2$: $n$-bits binary multiplicands, $P_{MA} = 0$:2 $n$-bits approximate product
2. Calculate $k_1$: leading one position of $N_1$
3. Calculate $k_2$: leading one position of $N_2$
4. Calculate $x_1$: shift $N_1$ to the left by $n-k_1$bits
5. Calculate $x_2$: shift $N_2$ to the left by $n-k_2$ bits
6. Calculate $k_{12} = k_1 + k_2$
7. Calculate $x_{12} = x_1 + x_2$
8. IF $x_{12} \geqslant 2^n$ (i.e. $x_1 + x_2 \geqslant 1$):
   (a) Calculate $k_{12} = k_{12} + 1$
   (b) Decode $k_{12}$ and insert $x_{12}$ in that position of $P_{approx}$
   ELSE:
   (a) Decode $k_{12}$ and insert '1' in that position of $P_{approx}$
   (b) Append $x_{12}$ immediately after this one in $P_{approx}$
9. Approximate $N_1\cdot N_2 = P_{MA}$
```

A step-by-step example illustrating Mitchell's algorithm-based multiplication is shown in Example 1.

Example 1 (*Mitchell's multiplication of 234 and 198).*

$$\begin{array}{ll} N_1 = 234 = 11101010, & N_2 = 198 = 11000110 \\ k_1 = 0111, x_1 = 11010100 & k_2 = 0111, x_2 = 10001100 \\ k_1 + k_2 = 1110 & \\ x_1 + x_2 = 101100000 \geqslant 2^8 & k_{12} = 1110 + 1 = 1111 \\ P_{MA} = 101100000000000 = 45056 & P_{true} = 46332 \\ E_r = \frac{P_{true} - P_{MA}}{P_{true}} = 2.754\% & \end{array}$$

The MA produces a significant error percentage. The relative error increases with the number of bits with the value of '1' in the mantissas. The maximum possible relative error for MA multiplication is around 11%, and the average error is around 3.8% [15,17]. The error in MA is always positive so it can be reduced by successive multiplications. Mitchell analyzed this error and proposed the following analytical expression for the error correction:

$$(N_1 \cdot N_2)_{MAC} = \begin{cases} P_{MA} + 2^{k_1+k_2}(x_1 \cdot x_2), & x_1 + x_2 < 1 \\ P_{MA} + 2^{k_1+k_2}(1 - x_1) \cdot (1 - x_2), & x_1 + x_2 \geqslant 1 \end{cases} \tag{6}$$

where $2^{k_1+k_2}(x_1 \cdot x_2)$ and $2^{k_1+k_2}(1 - x_1) \cdot (1 - x_2)$ are the correction terms proposed by Mitchell. To calculate the correction terms we have to:

1. calculate $x_1 \cdot x_2$ or $(1 - x_1) \cdot (1 - x_2)$ depending on $x_1 + x_2$ in the same way as described in (5),
2. scale the correction term by the factor $2^{k_1+k_2}$,
3. add the correction term to the product P_{MA}.

The error correction can be done iteratively and the error can be reduced to an arbitrary value. One important observation (from Algorithm 1) is that the error correction can start only after the term $x_1 + x_2$ is calculated.

Numerous attempts have been made to improve the MA's accuracy. Hall [7], for example, derived different equations for error correction in the logarithm and antilogarithm approximation in four separate regions, depending on the mantissa value, reducing the average error to 2%, but increasing the complexity of the realization. Abed and Siferd [1,2] derived correction equations with coefficients that are a power of two, reducing the error and keeping the simplicity of the solution. Among the many methods that use look-up tables for error correction in the MA algorithm, McLaren's method [16], which uses a look-up table with 64 correction coefficients calculated in dependence of the mantissas values, can be selected as one that has satisfactory accuracy and complexity. A recent approach for the MA error correction, reducing the number of bits with the value of '1' in mantissas by operand decomposition, was presented by Mahalingam and Rangantathan [15]. They proposed and implemented the Operand Decomposition-based Mitchell multiplier (OD-MA). The proposed OD-MA multiplier decreases the error percentage of the original MA multiplier by 44.7%, on average, but almost doubles the gates and power required when compared to the original MA multiplier.

3. Proposed solution

The proposed solution simplifies logarithm approximation introduced in (5) and introduces an iterative algorithm with various possibilities for achieving the multiplication error as small as required and the possibility of achieving the exact result. By simplifying the logarithm approximation introduced in (5), the correction terms could be calculated almost immediately after the calculation of the approximate product has been started. In such a way, the high level of parallelism can be achieved by the principle of pipelining, thus reducing the complexity of the logic required by (5) and increasing the speed of the multiplier with error correction circuits.

Looking at the binary representation of the numbers in (1), we can derive a correct expression for the multiplication:

$$\begin{aligned} P_{true} &= N_1 \cdot N_2 = 2^{k_1}(1 + x_1) \cdot 2^{k_2}(1 + x_2) \\ &= 2^{k_1+k_2}(1 + x_1 + x_2) + 2^{k_1+k_2}(x_1 x_2) \end{aligned} \tag{7}$$

To avoid the approximation error, we have to take into account the next relation derived from (1):

$$x \cdot 2^k = N - 2^k \tag{8}$$

The combination of (7) and (8) gives:

$$\begin{aligned} P_{true} &= (N_1 \cdot N_2) \\ &= 2^{(k_1+k_2)} + (N_1 - 2^{k_1})2^{k_2} + (N_2 - 2^{k_2})2^{k_1} + (N_1 - 2^{k_1}) \\ &\quad \cdot (N_2 - 2^{k_2}) \end{aligned} \tag{9}$$

Let

$$P^{(0)}_{approx} = 2^{(k_1+k_2)} + (N_1 - 2^{k_1})2^{k_2} + (N_2 - 2^{k_2})2^{k_1} \tag{10}$$

be the first approximation of the product. It is evident that

$$P_{true} = P^{(0)}_{approx} + (N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2}) \tag{11}$$

The proposed method is very similar to MA. The error is caused by ignoring the second term in (11). The term $(N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2})$ requires multiplication. If we discard it from (11), we have the approximate multiplication that requires only few shift and add operations. Computational equation to MA multiplier (5) requires the comparison of the addend $x_1 + x_2$ with 1. Instead of ignoring it and instead of approximating the product as proposed in (5), we can calculate the product $(N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2})$ in the same way as $P^{(0)}_{approx}$ and repeat the procedure until exact result is obtained. The evident difference between the proposed method and the method proposed by Mitchell is that the proposed method avoids the comparison of the addend $x_1 + x_2$ with 1. In such a way, the error correction can start immediately after removing the leading ones form the both input operands N_1 and N_2. This is a key factor that allows further pipelining and reduces the required gates as we will show lately. For this reason, an iterative calculation of the correction terms is proposed, as follows.

The absolute error after the first approximation is

$$E^{(0)} = P_{true} - P^{(0)}_{approx} = (N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2}) \tag{12}$$

Note that $E^{(0)} \geqslant 0$. The two multiplicands in (12) are binary numbers that can be obtained simply by removing the leading '1' in the numbers N_1 and N_2 so we can repeat the proposed multiplication procedure with these new multiplicands

$$E^{(0)} = C^{(1)} + E^{(1)} \tag{13}$$

where $C^{(1)}$ is the approximate value of $E^{(0)}$ and $E^{(1)}$ is an absolute error when approximating $E^{(0)}$. The combination of (11) and (13) gives

$$P_{true} = P^{(0)}_{approx} + C^{(1)} + E^{(1)} \tag{14}$$

We can now add the approximate value of $E^{(0)}$ to the approximate product P_{approx} as a correction term by which we decrease the error of the approximation

$$P^{(1)}_{approx} = P^{(0)}_{approx} + C^{(1)} \tag{15}$$

If we repeat this multiplication procedure with i correction terms, we can approximate the product as

$$P^{(i)}_{approx} = P^{(0)}_{approx} + C^{(1)} + C^{(2)} + \cdots + C^{(i)} = P^{(0)}_{approx} + \sum_{j=1}^{i} C^{(j)} \tag{16}$$

The procedure can be repeated, achieving an error as small as necessary, or until at least one of the residues becomes a zero. Then the final result is exact: $P_{approx} = P_{true}$. The number of iterations required for an exact result is equal to the number of bits with the value of '1' in the operand with the smaller number of bits with the value of '1'. The proposed iterative MA-based multiplication is given in Algorithm 2.

Algorithm 2 (Iterative MA-based Algorithm with *i* correction terms).

```
1. $N_1$, $N_2$: $n$-bits binary multiplicands, $P^{(0)}_{approx} = 0$ : $2n$-bits first approximation,$C^{(i)} = 0$: $2n$-bits $i$ correction terms, $P_{approx} = 0$: $2n$-bits product
2. Calculate $k_1$: leading one position of $N_1$
3. Calculate $k_2$: leading one position of $N_2$
4. Calculate $(N_1 - 2^{k_1})2^{k_2}$: shift $(N_1 - 2^{k_1})$ to the left by $k_2$ bits
5. Calculate $(N_2 - 2^{k_2})2^{k_1}$: shift $(N_2 - 2^{k_2})$ to the left by $k_1$ bits
6. Calculate $k_{12} = k_1 + k_2$
7. Calculate $2^{(k_1+k_2)}$: decode $k_{12}$
8. Calculate $P^{(0)}_{approx}$: add $2^{(k_1+k_2)}$, $(N_1 - 2^{k_1})2^{k_2}$ and $(N_2 - 2^{k_2})2^{k_1}$
9. Repeat $i$-times or until $N_1 = 0$ or $N_2 = 0$:
    (a) Set: $N_1 = N_1 - 2^{k_1}$, $N_2 = N_2 - 2^{k_2}$
    (b) Calculate $k_1$: leading one position of $N_1$
    (c) Calculate $k_2$: leading one position of $N_2$
    (d) Calculate $(N_1 - 2^{k_1})2^{k_2}$: shift $(N_1 - 2^{k_1})$ to the left by $k_2$ bits
    (e) Calculate $(N_2 - 2^{k_2})2^{k_1}$: shift $(N_2 - 2^{k_2})$ to the left by $k_1$ bits
    (f) Calculate $k_{12} = k_1 + k_2$
    (g) Calculate $2^{(k_1+k_2)}$: decode $k_{12}$
    (h) Calculate $C^{(i)}$: add $2^{(k_1+k_2)}$, $(N_1 - 2^{k_1})2^{k_2}$ and $(N_2 - 2^{k_2})2^{k_1}$
10. $P^{(i)}_{approx} = P^{(0)}_{approx} + \sum_i C^{(i)}$.
```

One of the advantages of the proposed solution is the possibility to achieve an arbitrary accuracy by selecting the number of iterations, i.e., the number of additional correction circuits, but more important is that the calculation of the correction terms can start immediately after removing the leading ones from the original operands, because there is no comparison of the sum of the mantissas with 1. The step-by-step example illustrating the proposed iterative algorithm multiplication with three correction terms is shown in Example 2. With three correction terms, in Example 2 the correct result is achieved.

Example 2 (*Proposed multiplication of 234 and 198 with three correction terms*).

1. Initialization:

$$N_1 = 234 = 11101010 \quad N_2 = 198 = 11000110$$
$$P_{true} = 46332$$

2. Calculate $P^{(0)}_{approx}$:

$$k_1 = 0111, N_1 - 2^{k_1} = 01101010 \qquad k_2 = 0111, N_2 - 2^{k_2} = 01000110$$
$$k_1 + k_2 = 1110$$
$$(N_1 - 2^{k_1})2^{k_2} = 011010100000000 \qquad (N_2 - 2^{k_2})2^{k_1} = 010001100000000$$
$$2^{(k_1+k_2)} = 100000000000000$$
$$P^{(0)}_{approx} = 1001100000000000 = 38912 \quad E^{(0)}_r = 16.014\%$$

3. Calculate $C^{(1)}$ and $P^{(1)}_{approx}$:

$$N^{(1)}_1 = 01101010 \qquad N^{(1)}_2 = 01000110$$
$$k^{(1)}_1 = 0110, N^{(1)}_1 - 2^{k^{(1)}_1} = 00101010 \quad k^{(1)}_2 = 0110, N^{(1)}_2 - 2^{k^{(1)}_2} = 00000110$$
$$k^{(1)}_1 + k^{(1)}_2 = 1100$$
$$\left(N^{(1)}_1 - 2^{k^{(1)}_1}\right)2^{k^{(1)}_2} = 101010000000 \quad \left(N^{(1)}_2 - 2^{k^{(1)}_2}\right)2^{k^{(1)}_1} = 000110000000$$
$$2^{\left(k^{(1)}_1+k^{(1)}_2\right)} = 1000000000000$$
$$C^{(1)} = 1110000000000 = 7168$$
$$P^{(1)}_{approx} = 46080 \qquad E^{(1)}_r = 0.543\%$$

4. Calculate $C^{(2)}$ and $P^{(2)}_{approx}$:

$$N^{(2)}_1 = 00101010 \qquad N^{(2)}_2 = 00000110$$
$$k^{(2)}_1 = 0101, N^{(2)}_1 - 2^{k^{(2)}_1} = 00001010 \quad k^{(2)}_2 = 0010, N^{(2)}_2 - 2^{k^{(2)}_2} = 00000010$$
$$k^{(2)}_1 + k^{(2)}_2 = 0111$$
$$(N^{(2)}_1 - 2^{k^{(2)}_1})2^{k^{(2)}_2} = 0101000 \qquad (N^{(2)}_2 - 2^{k^{(2)}_2})2^{k^{(2)}_1} = 1000000$$
$$2^{\left(k^{(2)}_1+k^{(2)}_2\right)} = 10000000$$
$$C_{(2)} = 11101000 = 232$$
$$P^{(2)}_{approx} = 46312 \qquad E^{(2)}_r = 0.043\%$$

5. Calculate $C^{(3)}$ and $P^{(3)}_{approx}$:

$$N_1^{(3)} = 00001010 \qquad N_2^{(3)} = 00000010$$
$$k_1^{(3)} = 0011, N_1^{(3)} - 2^{k_1^{(3)}} = 00000010 \quad k_2^{(3)} = 0001, N_2^{(3)} - 2^{k_2^{(3)}} = 00000000$$
$$k_1^{(3)} + k_2^{(2)} = 0111$$
$$(N_1^{(3)} - 2^{k_1^{(3)}})2^{k_2^{(3)}} = 100 \qquad (N_2^{(3)} - 2^{k_2^{(3)}})2^{k_1^{(3)}} = 000$$
$$2^{\left(k_1^{(3)}+k_2^{(3)}\right)} = 10000$$
$$C_{(3)} = 10100 = 20$$
$$P_{approx}^{(3)} = 46332 \qquad E_r^{(3)} = 0.0\%$$
$$P_{true} = P_{approx}^{(3)}$$

4. Hardware implementation

In order to evaluate the device utilization and the performance of the proposed multiplier, we implemented different multipliers on the Xilinx xc3s1500-5fg676 FPGA [28]. We implemented the 16-bit Mitchell's multiplier (MA), the 16-bit Operand-decomposition multiplier (OD-MA) and eight 16-bit proposed multipliers: a multiplier with no correction terms, three multipliers with one, two and three correction terms and a pipelined multiplier with no correction terms and three pipelined multipliers with one, two and three correction terms. The MA and OD-MA multipliers are implemented without pipelining as described in [17,15]. The proposed iterative multipliers are implemented as follows.

4.1. Basic block

A basic block (BB) is the proposed multiplier with no correction terms. The task of the basic block is to calculate one approximate product according to (10). The 16-bit basic block is presented in Fig. 1. This basic block consists of two leading-one detectors (LODs), two encoders, two 32-bit barrel shifters, a decoder unit and two 32-bit adders. Two input operands are given to the LODs and the encoders. The implementation of the 4-bit LOD unit is presented in Fig. 2. It is implemented with multiplexers 2-to-1 and 2-input AND gates.

The LOD units are used to remove the leading one from the operands, which are then passed to the barrel shifters. The LOD

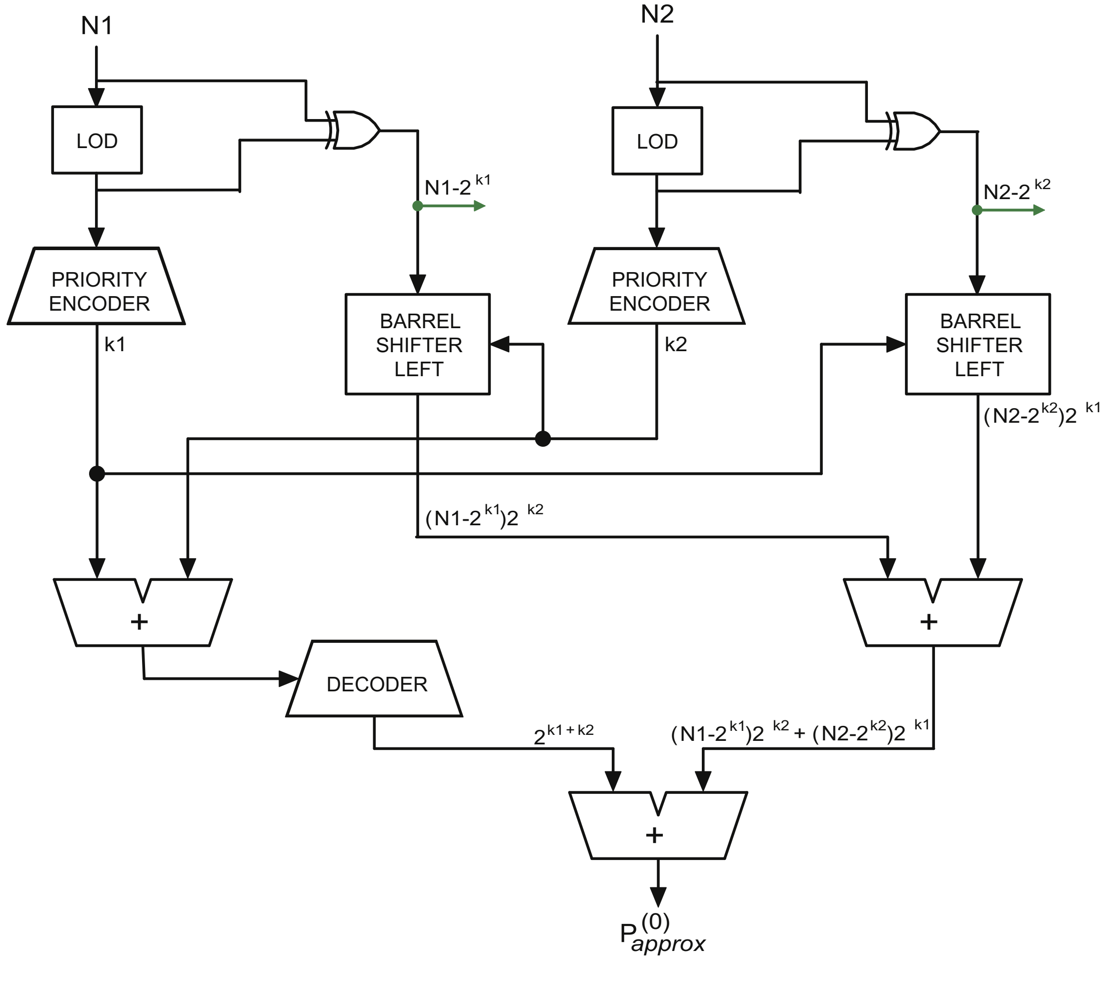


Fig. 1. Block diagram of a basic block of the proposed iterative multiplier.

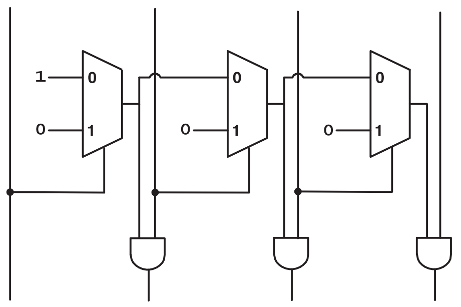


Fig. 2. 4-bit leading-one detector (LOD) circuitry.

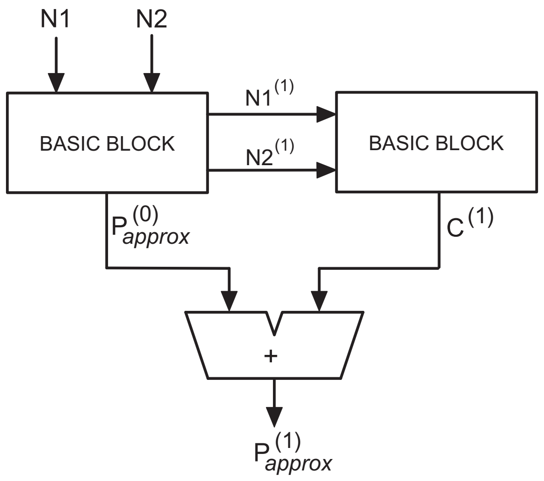


Fig. 3. Block diagram of the proposed multiplier with error-correction circuit.

units also include zero detectors, which are used to detect the zero operands. The LOD units and the zero detectors are implemented as in [1], while the barrel shifters are used to shift the residues according to (10). The decode unit decodes $k_1 + k_2$, i.e. it puts the leading one in the product. The leading one and the two shifted residues are then added to form the approximate product. The basic block is then used in subsequent implementations to implement correction circuits.

4.2. Implementation with correction circuits

To increase the accuracy of the multiplier, we implemented multipliers with error-correction circuits (ECC). The error-correction circuit is used to calculate the term $C^{(1)}$ in (14) and thus approximates the term $(N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2})$ in (9). To implement the proposed multipliers, we used the cascade of basic blocks. A block diagram of the proposed logarithmic multiplier with one error-correction circuit is shown in Fig. 3. The multiplier is composed of two basic blocks, of which the first one calculates the first approximation of the product $P^{(0)}_{approx}$, while the second one calculates the error-correction term $C^{(1)}$. We have implemented three multipliers: with one error-correction circuit, with two error-correction circuits and with three error correction circuits. Each correction circuit is implemented as a basic block and is used to approximate the product according to (16).

4.3. Pipelined implementation of the basic block

To decrease the maximum combinational delay in the basic block, we used pipelining to implement the basic block from Fig. 1. The pipelined implementation of the basic block is shown in Fig. 4 and has four stages. The stage 1 calculates the two characteristic numbers k_1, k_2 and the two residues $N_1 - 2^{k_1}$, $N_2 - 2^{k_2}$. The residues are outputted in stage 2, which also calculates $k_1 + k_2, (N_1 - 2^{k_1}) \cdot 2^{k_2}$ and $(N_2 - 2^{k_2}) \cdot 2^{k_1}$. The stage 3 calculates $2^{k_1+k_2}$ and $(N_1 - 2^{k_1}) \cdot 2^{k_2} + (N_2 - 2^{k_2}) \cdot 2^{k_1}$. The stage 4 calculates the approximation of the product $P^{(0)}_{approx}$.

4.4. Pipelined implementation with correction circuits

As the previous implementations with correction circuits show a substantial increase in combinational delay as each correction circuit is added, we used pipelining to implement the multiplier with error-correction circuits. Actually, the two basic blocks from Fig. 3 cannot really work in parallel in real-time, because the correction block cannot start until the residues are calculated from the first basic block. As in the pipelined implementation of the basic block the residues are available after the first stage, the correction circuit can now start to work immediately after the first stage from the prior block is finished. The pipelined multiplier with two correction circuits is presented in Fig. 5. The multiplier is composed of the three pipelined basic blocks, of which the first one calculates an approximate product $P^{(0)}_{approx}$, while the second and the third ones calculate the error-correction terms $C^{(1)}$ and $C^{(2)}$, respectively. The initial latency of the pipelined multiplier with two correction circuits is 6 clock periods, but after the initial latency, the products are calculated in each clock period. We have implemented three such pipelined multipliers: with one error-correction circuit, with two error-correction circuits and with three error-correction circuits.

4.5. Device utilization

For the design entry, we used the Xilinx ISE 11.3 – WebPACK [29] and designed with VHDL [32]. The design was synthesized with the Xilinx Xst Release 11.3 for Linux [30].

The device utilization (the number of slices, the number of 4-input look-up tables (LUTs) and the number of input–output blocks (IOBs)) for all ten implemented multipliers are given in Tables 1 and 2.

From Table 1 we can see that the Mitchell's multiplier (MA) requires almost 17% more logic than the basic block (BB). This is because MA requires additional logic to approximate product according to the addend $(x_1 + x_2)$. Also, it can be observed from Table 1 that OD-MA requires slightly more logic than BB with one ECC.

The maximum frequencies for the non-pipelined and pipelined implementations are given in Table 3.

4.6. Power analysis

To analyze the power consumptions in all eight multipliers we used the Xilinx XPower Analyzer 11.3 [31]. To increase the accuracy of the power analysis we synthesized all eight multipliers in the Xilinx xc3s1500-5fg676 FPGA [28] and assigned all I/O to pins. The power consumption is estimated at a clock frequency of 25 MHz with a signal (toggle) rate of 12.5%. With the Xilinx XPower Analyzer we have estimated the three main power components: quiescent power, logic and signals power and the IOBs power. Quiescent power (also referred to as leakage) is the power consumed by the FPGA powered on with no signals switching [31]. Quiescent power does not depend on the design programmed in the FPGA and is often referred to as power consumption of a non-programmed device. Typically, post-programmed quiescent power is equal or close to the pre-programmed quiescent power. Logic and signals power is average power consumption from the user

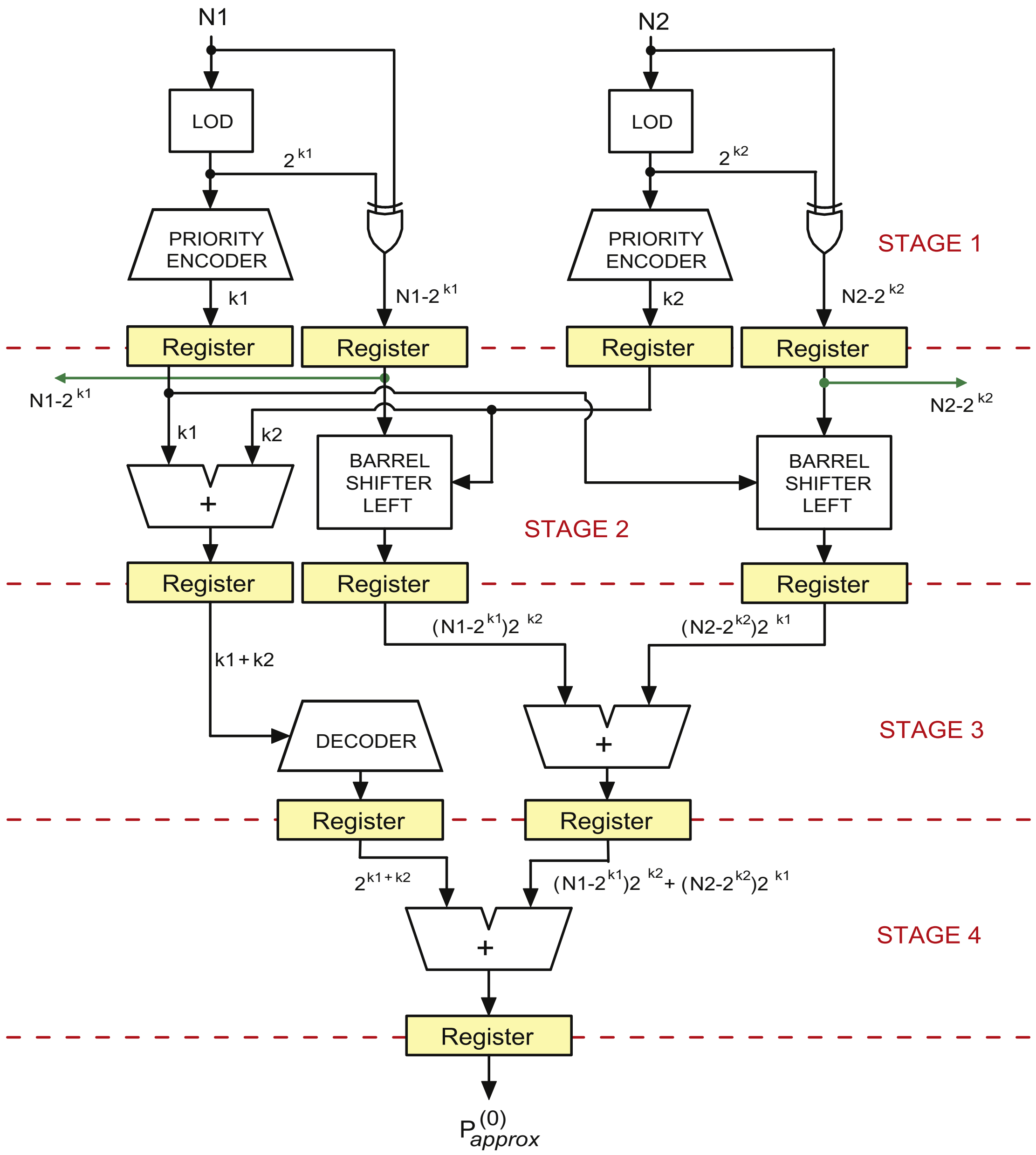


Fig. 4. Block diagram of a pipelined basic block of the proposed iterative multiplier.

logic utilization and switching activity. Logic power is typically under the designer's control and depends on the design being implemented (number of LUTs used), clock speed and signal rate.

The estimated power consumptions for the non-pipelined implementations are given in Table 4 and for the pipelined implementations are given in Table 5.

We can see that the total power consumption increases a little with the error-correction circuits added. This is because most of the total power consumption is due to quiescent power and the power consumed in I/O blocks. The power consumed in logic and signals is almost doubled with each correction circuit added, but represents only 1.5–6 % of the total power. Most DSP applications require a number of multipliers. For example, N-point convolution or correlation requires more than N^2 multipliers, where N is usually greater than 256. Therefore, it is of great importance to implement multiplier with low logic and signals power.

5. Error analysis

The relative error after the first approximation is given by

$$E_r^{(0)} = \frac{P_{true} - P_{approx}^{(0)}}{P_{true}} = \frac{E^{(0)}}{P_{true}} = \frac{(N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2})}{P_{true}} = \frac{2^{(k_1+k_2)} x_1 x_2}{2^{(k_1+k_2)}(1 + x_1 + x_2 + x_1 x_2)} = \frac{x_1 x_2}{(1 + x_1 + x_2 + x_1 x_2)} \quad (17)$$

where $0 \leqslant x_1, x_2 < 1$.

The relative error after approximating with i correction terms is a straightforward generalization of the above equation and is given by

$$E_r^{(i)} = \frac{E^{(i)}}{P_{true}} = \frac{\left(N_1^{(i)} - 2^{k_1^{(i)}}\right) \cdot \left(N_2^{(i)} - 2^{k_2^{(i)}}\right)}{P_{true}} \quad (18)$$

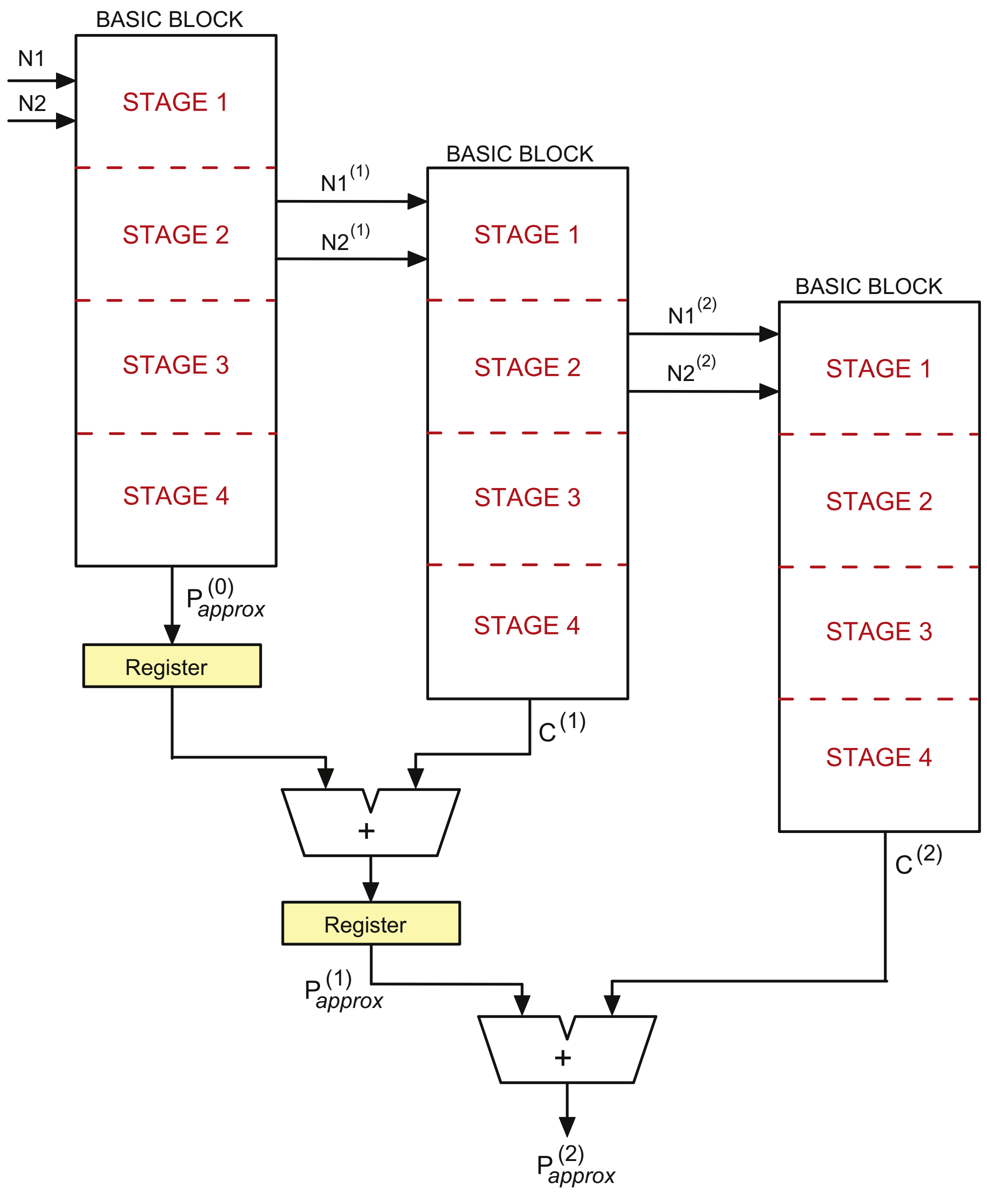


Fig. 5. Block diagram of the proposed pipelined multiplier with two pipelined error-correction circuits.

Table 1
Device utilization for the non-pipelined implementations.

Multiplier	4-input LUTs	Slices	Slice FFs	IOBs
MA	622	321	66	99
OD–MA	1187	604	101	99
BB	533	276	64	99
BB + 1 ECC	1099	564	80	99
BB + 2 ECC	1596	814	77	99
BB + 3 ECC	1937	993	78	99

Table 2
Device utilization for the pipelined implementations.

Multiplier	4-input LUTs	Slices	Slice FFs	IOBs
BB	404	216	170	99
BB + 1 ECC	803	427	306	99
BB + 2 ECC	1189	635	440	99
BB + 3 ECC	1546	824	569	99

Table 3
Maximum frequency for the non-pipelined and pipelined implementations.

Multiplier	Non-pipelined (MHz)	Pipelined (MHz)
MA	50.554	–
OD–MA	40.330	–
BB	58.075	153.335
BB + 1 ECC	50.180	153.335
BB + 2 ECC	41.429	153.335
BB + 3 ECC	37.826	153.335

Table 4
Estimated power consumption for the non-pipelined implementations.

Multiplier	Logic and signals (mW)	IO Blocks (mW)	Quiescent	Total (mW)
MA	5.15	45.68	151.29	202.12
OD–MA	9.06	31.39	151.28	191.73
BB	3.05	45.69	151.29	200.02
BB + 1 ECC	5.3	45.69	151.31	202.3
BB + 2 ECC	8.25	46.63	151.38	206.25
BB + 3 ECC	10.15	48.8	151.44	210.39

Table 5
Estimated power consumption at 25 MHz for the pipelined implementations.

Multiplier	Logic and signals (mW)	IO Blocks (mW)	Quiescent (mW)	Total (mW)
BB	3.72	51.26	152.06	207.04
BB + 1 ECC	7.32	51.62	152.66	211.6
BB + 2 ECC	10.66	51.67	153.03	215.36
BB + 3 ECC	13.37	51.93	153.42	218.72

Table 6
Maximum relative errors per number of ECCs used.

ECCs	$\boldsymbol{E}_{r,max}$ (%)
0	25.0
1	6.25
2	1.56
3	0.39
4	0.097
5	0.024

To show that the relative error decreases with adding the error-correction circuits, we have to prove that the absolute error in the current $(i+1)$th iteration is lower than the absolute error in the previous ith iteration, i.e.

$$E_r^{(i+1)} < E_r^{(i)} \tag{19}$$

Because we remove the leading ‘1’ from the multiplicands in each iteration, the following inequalities hold

$$k_1^{(i)} \leqslant k_1 - i, \quad k_2^{(i)} \leqslant k_2 - i \tag{20}$$

We can write

$$N_1^{(i)} = N_1 - 2^{k_1} - \sum_{j=1}^{i-1} 2^{k_1^{(j)}} \tag{21}$$

The same holds for $N_2^{(i)}$

$$N_2^{(i)} = N_2 - 2^{k_2} - \sum_{j=1}^{i-1} 2^{k_2^{(j)}} \tag{22}$$

From (20)–(22) we can write

$$N_1^{(i+1)} - 2^{k_1^{i+1}} < N_1^{(i)} - 2^{k_1^{i}} \tag{23}$$

and

$$N_2^{(i+1)} - 2^{k_2^{i+1}} < N_2^{(i)} - 2^{k_2^{i}} \tag{24}$$

Therefore,

$$E_r^{(i+1)} < E_r^{(i)} \tag{25}$$

If we assume the worst case when all the bits in the multiplicands are ‘1’, then the characteristic numbers are decreased by one in each iteration, i.e., the following holds

$$k_1^{(i)} = k_1 - i, \quad k_2^{(i)} = k_2 - i \tag{26}$$

Then the maximum relative error in the ith iteration is

$$\begin{aligned} E_{r,max}^{(i)} &= \frac{2^{(k_1-i+k_2-i)} \cdot x_1^{(i)} x_2^{(i)}}{2^{(k_1+k_2)} \cdot (1 + x_1 + x_2 + x_1 x_2)} \\ &= 2^{-2i} \cdot \frac{x_1^{(i)} x_2^{(i)}}{(1 + x_1 + x_2 + x_1 x_2)} \end{aligned} \tag{27}$$

It is obvious that the maximum relative error decreases exponentially with a ratio of at least 2^{-2i}, and it reaches 0 when one of the multiplicands is 0. Table 6 presents the maximum relative errors for different numbers of ECCs.

In order to evaluate the average relative error (AER), the proposed algorithm is applied to all combinations of n-bit non-negative numbers,and the average relative error is calculated from

$$AER = \frac{1}{N} \sum_{i=1}^{N} E_r \tag{28}$$

where N is the number of multiplications performed. For example, for 12-bit numbers, all the combinations of numbers ranging from 1 to 4095 are multiplied and the average relative error is calculated. The AER calculation is made in four cases: without an error-correction circuit, with one correction, with two corrections and with three corrections. The results from Table 7 represent the relative error rate for various cases. The results from Table 8 should be used when we want to decide how many error-correction circuits are necessary to achieve the desired average relative errors.

These results are compared with the results from [15], since it is the latest paper with a complete overview of the various solutions. Comparing the 8-bit and 16-bit average error percentages, we can see that our solution with only one error-correction circuits outperforms the OD-MA multiplier. The OD-MA multiplier has an average relative error between 2.07% and 2.15%, while the proposed multiplier with one error correction circuit has an average relative error between 0.83% and 0.99%.

6. Example

Let us consider two successive or near video frames, marked as the reference frame and the observed frame. For a block from observed frame (observed block), block matching techniques try to find the best matching block in the reference frame. When the best matching is found, the displacement is calculated and used as a motion vector, in applications like MPEG video compression [24]. For efficient compression, a compromise between the speed of the calculation and the accuracy of the motion vector is necessary.

Table 8
Average relative errors [%] for 0, 1, 2 and 3 correction terms.

No. bits	BB	BB + 1 ECC	BB + 2 ECC	BB + 3 ECC
8	8.9131	0.8337	0.0708	0.0048
12	9.3692	0.9726	0.1029	0.0106
16	9.4124	0.9874	0.1070	0.0117

Table 7
Relative error rate [%].

Error	8 bits			12 bits			16 bits		
	1 ECC	2 ECC	3 ECC	1 ECC	2 ECC	3 ECC	1 ECC	2 ECC	3 ECC
<0.1%	32.9	79.9	99.0	20.6	71.6	98.2	19.3	70.6	98.0
<0.5%	54.8	96.9	100	48.1	95.7	100	47.4	95.5	100
<1%	69.9	99.6	100	65.6	99.4	100	65.2	99.4	100

We considered the matching techniques based on a block correlation. If we denote the observed block with $F(i,j)$, where i and j are the pixels' coordinates, and a respective block in the reference frame with $S(i,j)$, assuming the block size is $N \times N$, then the correlation coefficients $C(x,y)$ are calculated for all positions (x,y) from the reference region as follows:

$$C(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} F(i,j) \cdot S(x+i, y+j) \tag{29}$$

The maximum value of the correlation coefficients determines the motion vector coordinate. The method is simple but computationally expensive, because it requires a large number of multiplications, especially when the block size is large.

Let us define a particular averaged error for the matching block, individually:

$$PAE = \frac{1}{N_C} \sum_{i=1}^{N_C} \frac{|C_i^a - C_i^t|}{C_i^t} \cdot 100\% \tag{30}$$

where N_C is the total number of correlation coefficients for one observed block regarding all possible block positions in the reference region, C_i^a is an approximated value of the correlation coefficient (the multiplication was performed with the proposed solution) and C_i^t is a true value of the correlation coefficient. The total averaged error was calculated as

$$TAE = \frac{1}{B} \sum_{i=1}^{B} PAE_i \tag{31}$$

where B is the total number of observed blocks in the reference region.

In order to show the usability of the proposed multiplier for motion-vector calculations, the described block-matching algorithm was applied on a selected region of the successive CT scan frames (Fig. 6). Observed blocks of dimensions 7×7 pixels were chosen from the reference region and the correlation coefficients were calculated for all positions in the reference region. The experiments were performed with the reference region size of 32×32 and 48×48 pixels.

The results obtained with the proposed algorithm (one and two correction terms) are compared with the results of the regular multiplication and shown in Table 9. A mismatch is defined as the percentage of false motion vectors obtained with the proposed multiplier. The TAE error is calculated and possibility of the wrong matching detection was examined.

From Table 9 we can conclude that the percentage of mismatching is very low. Due to the heavy computational requirements, the block matching is often performed in two stages, a rough estimation of a moving vector and then an accurate refinement [24]. In video compression, the errors in motion vectors will slightly decrease the compression. Since the proposed solution speeds up the compression algorithms, future investigation should address the influence of the motion vector errors on the level of compression.

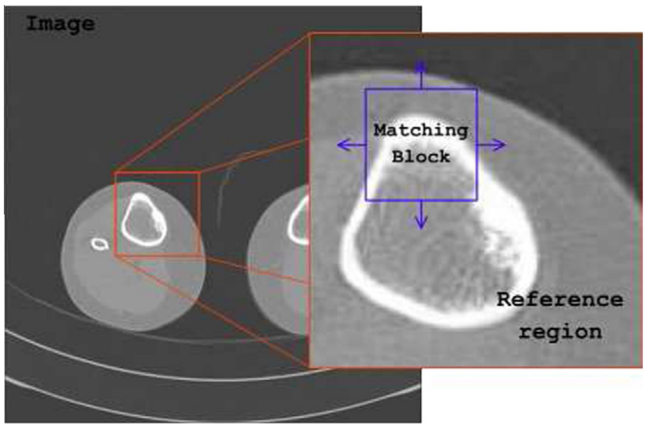


Fig. 6. Motion vector detection on successive CT scan frames.

Table 9
Error analysis for block matching algorithm.

Region	ECCs	TAE (%)	B	False vectors	Mismatching percentage (%)
32×32	1	0.548	676	27	3.99
32×32	2	0.034	676	1	0.15
48×48	1	0.566	1764	56	3.17
48×48	2	0.035	1764	10	0.56

7. Conclusions

In this paper, we have investigated and proposed a new approach to improve the accuracy and efficiency of Mitchell's algorithm-based multiplication. The proposed method is based on iteratively calculating the correction terms but avoids the comparison of the sum of mantissas with '1'. In such a way, the basic block for multiplication requires less logic resources for its implementation and the multiplier can achieve higher speeds.

We have shown that the calculation of the correction terms can be performed almost in parallel by pipelining the error correction circuits. After the initial latency, the products are calculated in each clock period, regardless the number of the error correction circuits used.

The proposed approach improves the relative average error and the error rate compared to the basic MA multiplication and OD-MA. The proposed multiplier with only one error correction circuit requires less logic resources than OD-MA and achieves smaller average relative error than OD-MA.

The proposed multiplier consumes significantly less logic and signals power than the MA multiplier. The power consumption for the proposed multiplier with one error correction circuit consumes only 58% of the logic and signals power consumed by OD-MA, while achieving notably smaller relative error and combinational delay than OD-MA.

The maximum combinational delay increases by 30–45% with each added correction circuit, but this was significantly improved by pipelining the four main stages in the basic block and pipelining the correction circuits.

Acknowledgments

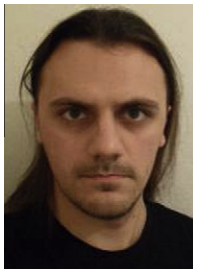
This project was funded by Slovenian Research Agency (ARRS) through Grants P2-0359 (National research program Pervasive computing) and BI-BA/10-11-026 (Bilateral Collaboration Project).

References

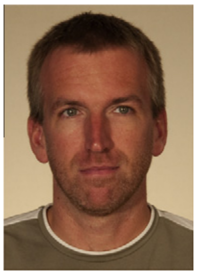
[1] K.H. Abed, R.E. Sifred, CMOS VLSI implementation of a low-power logarithmic converter, IEEE Transactions on Computers 52 (11) (2003) 1421–1433.
[2] K.H. Abed, R.E. Sifred, VLSI implementation of a low-power antilogarithmic converter, IEEE Transactions on Computers 52 (9) (2003) 1221–1228.
[3] L.V. Agostini, I.S. Silva, S. Bampi, Multiplierless and fully pipelined JPEG compression soft IP targeting FPGAs, Microprocessors and Microsystems 31 (8) (2007) 487–497.
[4] A. Daly, W. Marnane, T. Kerins, E. Popovici, An FPGA implementation of a GF(p) ALU for encryption processors, Microprocessors and Microsystems 28 (5–6) (2004) 253–260.
[5] S.M. Farhan, S.A. Khan, H. Jamal, An 8-bit systolic AES architecture for moderate data rate applications, Microprocessors and Microsystems 33 (3) (2009) 221–231.

[6] V. Gierenz, C. Panis, J. Nurmi, Parameterized MAC unit generation for a scalable embedded DSP core, Microprocessors and Microsystems 34 (5) (2010) 138–150.
[7] E.L. Hall, D.D. Lynch, S.J. Dwyer III, Generation of products and quotients using approximate binary logarithms for digital filtering applications, IEEE Transactions on Computers C-19 (2) (1970) 97–105.
[8] V. Hampel, P. Sobe, E. Maehle, Experiences with a FPGA-based reed/solomon-encoding coprocessor, Microprocessors and Microsystems 32 (5–6) (2008) 313–320.
[9] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, fourth ed., Morgan Kauffman Pub., 2007.
[10] H. Hinkelmann, P. Zipf, J. Li, G. Liu, M. Glesner, On the design of reconfigurable multipliers for integer and Galois field multiplication, Microprocessors and Microsystems 33 (1) (2009) 2–12.
[11] M-H. Jing, Z-H. Chen, J-H. Chen, Y-H. Chen, Reconfigurable system for high-speed and diversified AES using FPGA, Microprocessors and Microsystems 31 (2) (2007) 94–102.
[12] J.A. Kalomiros, J. Lygouras, Design and evaluation of a hardware/software FPGA-based system for fast image processing, Microprocessors and Microsystems 32 (2) (2008) 95–106.
[13] S.S. Kidambi, F. El-Guibaly, A. Antoniou, Area-efficient multipliers for digital signal processing applications, IEEE Transactions Circuits and Systems II: Analog and Digital Signal Processing 43 (2) (1996) 90–95.
[14] M.Y. Kong, J.M.P. Langlois, D. Al-Khalili, Efficient FPGA implementation of complex multipliers using the logarithmic number system, in: IEEE International Symposium on Circuits and Systems, ISCAS 2008, June 2008, pp. 3154–3157.
[15] V. Mahalingam, N. Rangantathan, Improving accuracy in Mitchell's logarithmic multiplication using operand decomposition, IEEE Transactions on Computers 55 (2) (2006) 1523–1535.
[16] D.J. Mclaren, Improved Mitchell-based logarithmic multiplier for low-power DSP applications, in: Proceedings of IEEE International SOC Conference 2003, 17–20 September 2003, pp. 53–56.
[17] J.N. Mitchell, Computer multiplication and division using binary logarithms, IRE Transactions on Electronic Computers EC-11 (1962) 512–517.
[18] H.T. Ngo, M. Zhang, L. Tao, V.K. Asari, Design of a high performance architecture for real-time enhancement of video stream captured in extremely low lighting environment, Microprocessors and Microsystems 33 (4) (2009) 273–280.
[19] C. Obimbo, B. Salami, A parallel algorithm for determining the inverse of a matrix for use in blockcipher encryption/decryption, The Journal of Supercomputing 39 (2) (2007) 113–130.
[20] G. Peretti, E. Romero, C. Marques, Testing digital low-pass filters using oscillation-based test, Microprocessors and Microsystems 32 (1) (2008) 1–9.
[21] M.H. Rais, Efficient hardware realization of truncated multipliers using FPGA, International Journal of Applied Science 5 (2) (2009) 124–128.
[22] S. Srot, A. Zemva, Design and implementation of the JPEG algorithm in integrated circuit, Electrotechnical Review 74 (4) (2007) 165–170.
[23] L-D. Van, C-C. Yang, Generalized low-error area-efficient fixed-width multipliers, IEEE Transactions Circuits and Systems I: Regular Paper 52 (8) (2005) 1608–1619.
[24] J. Watkinson, The MPEG Handbook: MPEG-1, MPEG-2, MPEG-4, second ed., Focal Press, 2004.
[25] A. Zemva, M. Verderber, FPGA-oriented HW/SW implementation of the MPEG-4 video decoder, Microprocessors and Microsystems 31 (5) (2007) 313–325.
[26] Y. Zhang, D. Chen, Y. Choi, L. Chen, S-B. Ko, A high performance ECC hardware implementation with instruction-level parallelism over $GF(2^163)$, Microprocessors and Microsystems 34 (6) (2010) 228–236.
[27] Y.Y. Zhang, Z. Li, L. Yang, S.W. Zhang, An efficient CSA architecture for montgomery modular multiplication, Microprocessors and Microsystems 31 (7) (2007) 456–459.
[28] Xilinx Inc. Spartan-3 FPGA Data Sheets, 2009 <http://www.xilinx.com/support/documentation/spartan-3_data_sheets.htm>.
[29] Xilinx ISE WebPACK Design Software, 2010 <http://www.xilinx.com/tools/webpack.htm>.
[30] Xilinx Inc. Xilinx Synthesis Technology (XST), 2009 <http://www.xilinx.com/tools/xst.htm>.
[31] Xilinx Inc. Xilinx XPower, 2009 <http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm>.
[32] EDA Industry Working Groups, 2010 <http://www.vhdl.org>.

Zdenka Babic received her B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering, University of Banja Luka, Bosnia and Herzegovina, in 1983, 1990 and 1999 respectively. She is an Associate Professor at the same faculty. Her main research interests are digital signal processing, image processing, circuits and systems. She is a member of the IEEE Signal Processing Society and IEEE Circuits and Systems Society.



Aleksej Avramovic received his B.Sc. degree in electrical engineering from the Faculty of Electrical Engineering, University of Banja Luka, Bosnia and Herzegovina, in 2007. He is a Teaching Assistant at same faculty. His research interests include digital signal processing and digital image processing. He is a student member of IEEE.



Patricio Bulic received his B.Sc. degree in electrical engineering, and M.Sc. and Ph.D. degrees in computer science from the University of Ljubljana, Slovenia, in 1998, 2001 and 2004, respectively. He is an Assistant Professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests include computer architecture, digital design, parallel processing and vectorization techniques. He is a member of the IEEE Computer Society and ACM.