A COMPARATIVE STUDY OF VECTOR DOT PRODUCT IMPLEMENTATIONS ON FPGAS

WITH DISTRIBUTED ARITHMETIC AND RESIDUE NUMBER SYSTEM
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Distributed Arithmetic</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Derivation of DA formulation</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Hardware organization</td>
<td>5</td>
</tr>
<tr>
<td>2.3 LUT Decomposition and size reduction techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Some estimates for an FPGA implementation</td>
<td>10</td>
</tr>
<tr>
<td>3. Residue Number system</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Basic description</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Implementation of RNS operators</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 Implementing a RNS adder</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Implementing a RNS multiplier</td>
<td>12</td>
</tr>
<tr>
<td>3.2.3 A MAC implementation using modulo adders and index calculus multipliers</td>
<td>13</td>
</tr>
<tr>
<td>3.3 Scaling and conversion issues</td>
<td>15</td>
</tr>
<tr>
<td>4. Combination of DA and RNS techniques</td>
<td>16</td>
</tr>
<tr>
<td>5. Description of proposed Implementation</td>
<td>18</td>
</tr>
<tr>
<td>5.1 Comparison methodology</td>
<td>18</td>
</tr>
<tr>
<td>5.2 Differences from earlier implementation proposal</td>
<td>19</td>
</tr>
<tr>
<td>Conclusion</td>
<td>19</td>
</tr>
<tr>
<td>Bibliography</td>
<td>19</td>
</tr>
</tbody>
</table>
Table of Figures

Figure 1: 1BAAT DA Hardware.............................................5
Figure 2: 2BAAT hardware with a single LUT.....................6
Figure 3: 2 BAAT implementation with replication of LUTs and processing the bits into two halves.................................................................7
Figure 4: 2 BAAT implementation with LUT replication and separate processing of odd and even bits.................................................................8
Figure 5: N BAAT full parallel implementation wiht N LUTs and a tree of parallel adders 9
Figure 6: Decomposition of a single LUT in figure 1 into two smaller LUTs 9
Figure 7: Index Calculus Multiplier ......................................12
Figure 8: MAC Architecture for an Arbitrary Moduli Set 14
Figure 9: MAC channels for (a) GF(p), (b) Z_2^m and (c) Z_p^m 14
Figure 10: Binary to modulo-m RNS conversion for a 16-bit input, using four-table partitioning .................................................................15
Figure 11: RNS to binary conversion using the CRT reconstruction using LUTs and a modulo adder tree. .................................................................16
Figure 12: Design of an RNS-DA Scaling accumulator....18
A Comparative Study of Vector Dot Product Implementations on FPGAs

USING DISTRIBUTED ARITHMETIC AND RESIDUE NUMBER SYSTEM

ABSTRACT

Vector dot product or multiply accumulate (MAC) is a frequently used operation in digital signal processing algorithms. When implementing digital signal processors (DSPs) using field programmable gate arrays (FPGAs), significant area and speed advantages are possible if the implementation architecture has a direct mapping on the underlying structure of FPGAs. Such a synergy is present in both distributed arithmetic (DA) and residue number system (RNS) based implementations. Both these techniques use FPGA look-up tables (LUTs) for implementing ROMs for their basic processing elements. In addition, RNS helps to hide the inefficiencies of FPGAs in handling large word lengths by breaking up the computation into parallel operations on smaller residues. The fusion of DA and RNS is also possible which combines the advantages of both the techniques.

1. INTRODUCTION

FPGAs are increasingly being used as an option for custom hardware designing for digital signal processing. However, compared to other technologies for custom hardware implementation (e.g. standard cells), FPGAs suffer from speed and area inefficiencies due to their flexible fine-grained LUT based architecture and a huge switch based routing network. The routing area and routing delays of FPGAs contributes largely to the area and delay inefficiency. When implementing DSPs on FPGAs, multipliers are required. Implementing large word length multipliers in FPGAs is area and speed inefficient. Nevertheless, FPGAs offer potential for parallelism that can be exploited if the architectures are tuned to the underlying structure of FPGAs. One such technique is distributed arithmetic. The use of distributed arithmetic for implementing custom DSPs was popular with the research community even before the advent of FPGAs because of its area efficient and relatively high-speed implementations. Distributed arithmetic replaces the multiplication with table look-ups and as such, it is very attractive for custom DSP implementation on FPGAs. Another technique is the use of residue number system (RNS) based implementations (modulo arithmetic). RNS offers parallelism by allowing operations such as addition/subtraction and multiplication to be performed using parallel, smaller residues. This results in faster adders with shorter carry propagation chains and facilitates the implementation of multiplication using LUTs since the operand sizes are small. In addition, the routing required to implement small word lengths is relatively efficient in FPGAs. DA and RNS techniques can be combined, resulting in area and speed advantages of both.

In this report, these two techniques are described and a description of hardware for implementation of vector dot product on FPGAs is presented. This is followed by details of methodology used for the comparison of implementation results of both techniques. The organization of this report is as follows. In section 2, the DA technique is described followed by a description of RNS approach in section 3. Section 4 discusses the fusion of DA and RNS. In Section 5, a description of the implementation strategy and methodology is given and finally conclusion follows.
2. DISTRIBUTED ARITHMETIC

Distributed arithmetic is a bit level rearrangement of a multiply accumulate to hide the multiplications [Andraka]. This re-arrangement is the basis of hardware reduction in a parallel multiply accumulate organization.

2.1 DERIVATION OF DA FORMULATION

The multiply accumulate operation is given by equation 1.

\[ y = \sum_{k=1}^{K} A_k x_k \]

Equation 1: Multiply accumulate

In equation 1, \(A_k\) are constants which depend upon a given DSP application and \(x_k\) are the inputs. In this way, equation 1 requires \(K\) product terms to be calculated and summed. Let \(x_k\) be of \(N\)-bits and in 2’s complement form. Equation 2 gives the representation of \(x_k\).

\[ x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \]

Equation 2: 2’s complement representation of \(x_k\)

Note that \(x_k\) is scaled, such that \(|x_k| < 1\) and \(b_{k0}\) is the sign bit. Substituting equation 2 in 1 gives:

\[ y = \sum_{k=1}^{K} A_k \left[ -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \right] \]

\[ y = -\sum_{k=1}^{K} \left( b_{k0} \cdot A_k \right) + \sum_{n=1}^{N-1} \sum_{k=1}^{K} \left( A_k \cdot b_{kn} \right) 2^{-n} \]

Equation 3: Substituting 2 in 1

Interchanging the order of the summations in equation 3 gives:

\[ y = -\sum_{k=1}^{K} A_k \cdot \left( b_{k0} \right) + \sum_{n=1}^{N-1} \sum_{k=1}^{K} \left( A_k \cdot b_{kn} \right) 2^{-n} \]

Equation 4: Interchanging the order of summations

Let, for \(n=0, 1, 2, \ldots, N-1\), we define a function \(f_n\) such that:

\[ f_n(b_{k0}b_{2n} \cdots b_{kn}) = \left[ \sum_{k=1}^{K} A_k b_{kn} \right] \]

Equation 5: Defining a function \(f_n\)
Substituting 5 in 4 gives:

\[ y = -f_0 + \sum_{n=1}^{N-1} f_n 2^{-n} \]

Equation 6: Expressing the equation in terms of \( f_n \)

Note that \( f_n(b_{1n} b_{2n} \ldots b_{Kn}) \) has only \( 2^K \) possible values that can be pre-calculated and stored in a look-up table of \( 2^K \) words addressed by the \( K \)-bits \( b_{1n} b_{2n} \ldots b_{Kn} \). Thus, the multiply accumulate can be calculated by \( N \) table look-ups, shifts and adds/subtract operations. This is the principle of distributed arithmetic approach to multiply accumulate calculation.

### 2.2 HARDWARE ORGANIZATION

In this section, the various hardware organizations for mechanizing the distributed arithmetic techniques are presented [Mintzer]. In its simplest form, DA is a bit-serial technique. The basic hardware configuration for one bit at a time (1BAAT) or serial operation of DA is shown in figure 1.

![Figure 1: 1BAAT DA Hardware](image)

The hardware in figure 1 is for \( K=8 \). If it is assumed that \( N=8 \), it will require 8 clock cycles to perform the multiply accumulate. The last cycle always involves a subtract operation instead of an add operation. The shifter shifts the bits 1 times during each cycle.

Let us define each the output of each look-up operation by \( \text{sum}[i] \) such that for the example given in figure 1, the first table look-up output is \( \text{sum}[7] \) followed by \( \text{sum}[6], \text{sum}[5] \) and the last output which is subtracted is \( \text{sum}[0] \). In this way, equation 6 can be rephrased as

\[ y = -(\text{sum}[0]) + (\text{sum}[1])2^{-1} + (\text{sum}[2])2^{-2} + \cdots + (\text{sum}[N-1])2^{-(N-1)} \]

Equation 7: Rephrasing equation 6 in terms of LUT outputs
In order to increase the number of bits processed in a cycle from more than 1 bit at a time and thus increase the throughput, more memory (a larger memory of more LUTs) can be employed. In general, an implementation processing L bits at a time will require \( N/L \) cycles to complete. Depending upon whether we use a larger LUT or more of smaller LUTs, 3 possible hardware configurations for 2 BAAT DA operation are shown in figure 2, figure 3 and figure 4.

In figure 2, the memory requirements have grown to \( 2^{2K} \) to support 2 BAAT operation. The input registers are now shifted two at time. Each LUT output is given by:

\[
\left( \text{sum}[n] + \text{sum}[n + 1] \right) 2^{-1}
\]

This introduces some complexities because the \( \text{sum}[0] \) has to be subtracted as before. This can be handled by grouping the terms as follows:

\[
y = -(\text{sum}[0]) + \left( \text{sum}[1] + (\text{sum}[2])2^{-1} \right) 2^{-1} + \cdots + \left( \text{sum}[N - 3] + (\text{sum}[N - 2])2^{-1} \right) 2^{-1} + (\text{sum}[N - 1])2^{-(N-1)}
\]

This implies that the first and the last terms are to handled in a different way. This is achieved by inserting two dummy bits in each shift register: one after the least significant bit and second after the most significant bit. Now the LUT outputs can be handled in the normal way with the last output being subtracted as usual. Note that this approach requires \( N/L + 1 \) cycles. Besides this, the memory requirements have grown exponentially and so this approach is often not realizable for a large \( K \) value.

The 2 BAAT approaches shown in figure 3 and 4 use the same principle of LUT duplication but differ in the way they group the LUT outputs. The implementation in figure 3 uses the following grouping:

\[
y = \left( - (\text{sum}[0]) + (\text{sum}[1])2^{-1} + (\text{sum}[2])2^{-2} + (\text{sum}[3])2^{-3} \right) + \left( (\text{sum}[4]) + (\text{sum}[5])2^{-1} + (\text{sum}[6])2^{-2} + (\text{sum}[7])2^{-3} \right) 2^{-4}
\]
This requires two scaling accumulators and post-accumulator adder, which is of double precision. Compared to this, the implementation in figure 4 requires one single precision pre-accumulator adder and a single scaling accumulator and uses the following grouping for terms:

\[
y = \left( - (\text{sum}[0]) + (\text{sum}[1])2^{-1} \right) + \left( (\text{sum}[2]) + (\text{sum}[3])2^{-2} + (\text{sum}[4]) + (\text{sum}[5])2^{-4} + (\text{sum}[6]) + (\text{sum}[7])2^{-6} \right)
\]

In this way, the implementation of figure 4 is more economical for the same throughput. For an N BAAT full parallel implementation, N LUTs are required with the LUT outputs being summed in a tree of adders without any scaling accumulators. The implementation is shown in figure 5 and is based on the grouping of terms as follows:
Figure 4: 2 BAAT implementation with LUT replication and separate processing of odd and even bits

\[
y = \left\{- (sum[0]) + (sum[1])2^{-1}\right\} + \left\{(sum[2]) + (sum[3])2^{-1}\right\}2^{-2} + \\
\left\{\left([sum[4]] + (sum[5])2^{-1}\right) + (sum[6]) + (sum[7])2^{-1}\right\}2^{-4}
\]

In this way, DA offers the classic speed-area tradeoff with increasing amount of parallelism.
2.3 LUT DECOMPOSITION AND SIZE REDUCTION TECHNIQUES

In a given hardware, an arbitrary size LUT implementation may not be possible. Such is the case even in FPGAs where the LUTs available are of uniform sizes. DA allows the flexibility...
of decomposing a large LUT in to smaller ones at the expense of the addition of parallel adders. For the example shown in figure 1, if the LUTs available are of 2xM bits, then an implementation of figure 6 can be used. Note that as much decomposition is performed, the adder stages will increase accordingly. Thus, if a 2xM bits LUT is used, there would be three parallel adders in a tree configuration before the scaling accumulator.

The sizes of LUTs are a major factor in the speed and area efficiency of DA based implementations. The size of LUTs can be reduced by a factor of 2 by using a technique described in [Stanley]. The technique uses a so-called 'offset binary' representation for address bits (the shifted values) of LUTs. This requires some additional preprocessing circuitry in the form of XOR gates before the LUT.

### 2.4 Some Estimates for an FPGA Implementation

Table 1 gives the estimates for a Xilinx XC4000 series FPGA CLBs requirement for implementing MACs with varying K values for N=16-bits and M=16 bits.

**Table 1: FPGA Resource Requirements for K=4, 6, 8 and 10 with N=16 and M=16**

<table>
<thead>
<tr>
<th>K</th>
<th>2^k</th>
<th>2^k x16</th>
<th>2^k x16/CLB</th>
<th>2^2k</th>
<th>2^2k x16/CLB</th>
<th>2 x2^k</th>
<th>2 x2^k x16</th>
<th>2 x2^k x16/CLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16</td>
<td>256</td>
<td>8</td>
<td>256</td>
<td>4096</td>
<td>128</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>1024</td>
<td>32</td>
<td>4096</td>
<td>65536</td>
<td>2048</td>
<td>128</td>
<td>2048</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>4096</td>
<td>128</td>
<td>65536</td>
<td>-</td>
<td>32677</td>
<td>512</td>
<td>8192</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>16384</td>
<td>512</td>
<td>2M</td>
<td>-</td>
<td>1M</td>
<td>2048</td>
<td>32768</td>
</tr>
</tbody>
</table>

It is evident from table 1 that using a single memory for 2 BAAT implementations results in exponential increase in resources. This will complicate routing and hence result in obviously slower designs than the LUT duplicating approach to 2 BAAT processing.

### 3. Residue Number System

The fundamental quality of RNS is ultra-high speed multiplication and addition, which comes from parallel processing of arithmetic operations [Taylor84]. RNS is a “carry-free” system that performs addition, subtraction and multiplication as concurrent (parallel) operations, sidestepping one of the principal arithmetic delays - managing carry information. Hence, RNS-based systems are preferable to conventional ones in view of the parallelism in such arithmetic operations. Since these operations are the only ones involved in a multiply accumulate, RNS can serve to dramatically accelerate it. Many innovative techniques have been used in the literature for the design of modulo adders and multipliers by making use of the properties of RNS and the number-theoretical properties of finite fields. Besides these techniques, the simplest approach is to use look-up table based operations because of the small operand sizes permitted by the RNS.

#### 3.1 Basic Description

A residue number system is characterized by a base that is not a single radix but an n-tuple of integers (m₁, m₂, ..., mₙ). Each of these mᵢ, i = 1, ..., n is called a modulus and is selected such that they are pair-wise relatively prime, i.e., GCD(mᵢ, mⱼ) = 1, for i ≠ j. Any integer X is represented in the RNS by an n-tuple of integers (x₁, x₂, ..., xₙ), where each xᵢ is a non-negative
integer satisfying the relationship \( X = m_i \cdot q_i + x_i \). Here, the quotient \( q_i \) is the largest integer such that \( 0 \leq x_i \leq (m_i - 1) \). The remainder \( x_i \) is hence the residue of \( X \) modulo \( m_i \), and the notations \( X \mod m_i \) and \( \text{im} \) are commonly used.

It follows from the Chinese Remainder Theorem (CRT) that for any given \( n \)-tuple satisfying the above relationships, there exists one and only one integer \( X \) such that \( 0 \leq X < M \) where \( M = \prod_{i=0}^{n-1} m_i \). The number \( X \) can be reconstructed from the \( n \)-tuple \( (x_1, x_2, \ldots, x_n) \) using the CRT equation, \( X = \left\lfloor \sum_{i=1}^{n} \frac{M}{m_i} a_i x_i \right\rfloor \) where \( a_i \) is defined by the relationship \( \frac{M}{m_i} a_i \equiv 1 \mod m_i \).

An alternate method, called mixed-radix-conversion (MRC), employs a mixed-radix-representation \( (a_1, a_2, \ldots, a_n) \) for a number \( X \), where \( X = \sum_{k=1}^{n} a_k \prod_{i=1}^{k-1} m_i \), with the digits satisfying \( 0 \leq a_i < m_i \) and \( i = 1, 2, \ldots, n \). Here, \( \prod_{i=1}^{n} m_i = 1 \).

Arithmetic operations in RNS are defined by:

\[
(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n) \otimes (\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_n) = (\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_n) \quad (1)
\]

with, \( \mathbf{z}_i = (\mathbf{x}_i \otimes \mathbf{y}_i) \mod m_i \), where, \( \otimes \) denotes any of the modulo operations of addition, subtraction or multiplication. The above definition has many implications. Firstly, the defined arithmetic operations are performed on smaller residues instead of the large number. In addition, these operations can be done simultaneously in all the moduli channels thereby speeding up the whole operation. Second, it gives fault isolation between different modulo channels. The net result is high-speed parallel arithmetic processing.

The basic operation involved in the Multiply-Accumulate Unit (MAC) is the summation of a number of products given by the general expression

\[
Y = \sum_{k=1}^{K} A_k X_k
\]

If \( A_k \) and \( X_k \) are represented in residue form using a moduli set \( \{m_1, m_2, \ldots, m_n\} \) then, as \( X_k = \{x_{k1}, x_{k2}, \ldots, x_{kn}\} \) and \( A_k = \{a_{k1}, a_{k2}, \ldots, a_{kn}\} \) in residue notation, the above summation thus becomes \( n \) independent summations:

\[
y_1 = \sum_{k=1}^{K} a_k x_{k1}, \quad y_2 = \sum_{k=1}^{K} a_k x_{k2}, \quad \ldots, \quad y_n = \sum_{k=1}^{K} a_k x_{kn}
\]

These summations correspond to the sum tuple, \( Y = \{y_1, y_2, \ldots, y_n\} \).

### 3.2 IMPLEMENTATION OF RNS OPERATORS

In order to exploit RNS parallelism, arithmetic units that efficiently implement the modular statement \( z_i = (\mathbf{x}_i \otimes \mathbf{y}_i) \mod m_i \) must be available. An alternate is to use look-up tables which are programmed to output the value \( (\mathbf{x}_i \otimes \mathbf{y}_i) \mod m_i \) upon the receipt of address \( [x_1 : y_1] \), which denotes the concatenation of \( x_1 \) and \( y_1 \). If \( m_i \) is bounded by \( 2^n \) for all \( i \), then the concatenated address is \( 2n \) bits wide. Therefore, RNS arithmetic can be performed as a look-up table mapping, provided the process does not exceed the address space of available memory units. Techniques for implementing RNS adder and multiplier have been documented as well. These implementations are described below.
3.2.1 IMPLEMENTING A RNS ADDER

Considering that it is possible to reach high speed implementations of ripple-carry adders of small word lengths by exploiting the dedicated structures in FPGAs that provide fast carry propagation between logic elements, in [Luiz et.al], authors have implemented RNS binary adders based on the work of [Dugdale]. Each adder modulo \( m \) follows the rules:

\[
(x + y) \mod m , \text{ where } x, y < m ,
\]

result = \( x + y \), if \( x + y < m \), or

= \( x + y - m \), if \( x + y \geq m \).

Two adders modulo \( 2^n \), where \( n = \lceil \log_2 m \rceil \) number of operands bits, are used to perform additions:

\[
\text{suma} = x + y, \text{ and } \text{sumb} = \text{suma} + (2^n - m);
\]

The correct result of \((x + y) \mod m\), is dependent on carries originated by \( \text{suma} \) and \( \text{sumb} \). Dugdale showed that if a carry occurs in any addition, the correct result is \( \text{sumb} \), otherwise \( \text{suma} \).

3.2.2 IMPLEMENTING A RNS MULTIPLIER

The index calculus approach to RNS multiplier is based on the properties associated to the Galois field theory [Radhakrishnan] [Jullien]. Given a Galois field \( GF(p) \), where \( p \) is a prime number, it is possible to generate all nonzero elements by using a primitive root of \( p \).

Let \( \alpha \) be a primitive root of \( p \), the nonzero elements \( \{1, 2, \ldots, p-1\} \) are generated by \( |\alpha^i|_p \), where the indexes, \( i \), are \( \{0, 1, \ldots, p-2\} \). The nonzero elements form an multiplicative group with multiplication modulo \( p \), and the index elements form an additive group with addition modulo \( (p-1) \). The Galois field multiplier is obtained by using the isomorphism between these two groups. Given \( x \) and \( y \), nonzero elements of \( GF(p) \), the result of multiplying \( x \) by \( y \), modulo \( p \), is:

\[
|x \cdot y|_p = \alpha^{i_x + i_y} \mod (p-1).
\]

Hence, this approach requires 3 steps to perform multiplication: (i) find the operand’s index, (ii) adding them in modulo \( p-1 \), and (iii) performing the inverse index operation. Implementations of index calculus multipliers are shown in figure 7. This RNS multipliers can be built with pipelined binary adders modulo \( m \) described in Section 3.2.1.

![Diagram of Index Calculus Multiplier](image-url)
In [Preethy], RNS MAC architecture is presented which uses the index-transform based approach as described above. The relatively prime moduli in any arbitrary moduli set take any of the three forms: \( p, 2^m, \) and \( p^m, \) or a factorable modulus with any of these factors, where \( p \) is prime and \( m \) is any integer. It follows from number theory that the groups formed by \( p, 2^m, \) and \( p^m \) integer elements fall into the category of Galois Field \( \text{GF}(p) \), and the integer rings \( \mathbb{Z}_{2^m} \) and \( \mathbb{Z}_{p^m} \) respectively. For the prime modulus \( p, \) the normal index mapping in \( \text{GF}(p) \) is used as already presented. In the case of finite integers, the procedure for finding an index set for the elements of \( \mathbb{Z}_{p^m} \) is given in [Vinogradov]. Using this, any integer \( X \in \{1,2,\ldots,2^m-1\} \) can be coded using a triplet index code \( <\alpha,\beta,\gamma> \) with the relationship

\[
X = 2^\alpha \left| 5^\beta (-1)^\gamma \right|_{2^m}
\]

where \( x \in \{0,1,\ldots,m-1\}, \beta \in \{0,1,\ldots,2^{m-2}-1\} \) and \( \gamma \in \{0,1\} \). Multiplication of two integers can now be carried out as follows: let \( X_1, X_2 \in \mathbb{Z}_{2^m}, X_1 \neq 0, X_2 \neq 0, \)

\[
X_1 = 2^\alpha_1 \left| 5^\beta_1 (-1)^\gamma_1 \right|_{2^m}
\]
\[
X_2 = 2^\alpha_2 \left| 5^\beta_2 (-1)^\gamma_2 \right|_{2^m}
\]

then,

\[
\left| X_1 X_2 \right|_{2^m} = 2^{\alpha_1+\alpha_2} \left| 5^{\beta_1+\beta_2} (-1)^{\gamma_1+\gamma_2} \right|_{2^m}
\]

The indices are added subject to the following constraints: \( \beta_1 \) and \( \beta_2 \) are added mod \( 2^{m-2} \), \( \gamma_1 \) and \( \gamma_2 \) are added mod 2, and \( z_1 \) and \( z_2 \) are added in normal binary mode. When the sum of \( z_2 \) equals \( m-1 \) the corresponding \( \beta \) and \( \gamma \) are made zero, and when it exceeds \( m-1 \), the final result is made zero. Thus, by storing the index and inverse index tables as described before, multiplication of the nonzero elements can be replaced by index addition. Similarly, in the case of \( \mathbb{Z}_{p^m} \), where \( p \) is odd, an index pair coding \( (\alpha, \beta) \) is used, where \( X \) is given by \( X = (g^\alpha \cdot p^\beta) \mod p^m \). The product of two numbers, \( (X_1 \cdot X_2) \mod p^m \) can now be calculated as follows: let \( X_1, X_2 \in \mathbb{Z}_{p^m}, X_1 \neq 0, X_2 \neq 0, \)

\[
X_1 = (g_1^\alpha \cdot p_1^\beta_1) \mod p^m, \quad X_2 = (g_2^\alpha_2 \cdot p_2^\beta_2) \mod p^m
\]

then \( (X_1 \cdot X_2) \mod p^m \) is given by:

\[
|X_1 \cdot X_2|_{p^m} = (g_1^\alpha_1 \cdot p_1^\beta_1 \cdot g_2^\alpha_2 \cdot p_2^\beta_2) \mod p^m = (g_1^{\alpha_1+\alpha_2} \cdot p_1^{\beta_1+\beta_2}) \mod p^m
\]

The indices are added subject to the following constraints: \( \alpha_1 \) and \( \alpha_2 \) are added mod \( \varphi(p^m) \), and \( \beta_1 \) and \( \beta_2 \) are added in normal binary mode. When the sum of \( \beta \) exceeds \( m-1 \), the final result is made zero. From the above, it may be noted that index transform techniques are suitable for any arbitrary moduli set subject to the only provision of relative primality. Thus, all the MAC channels can be designed using the index calculus techniques. The corresponding architecture consists of \( n \) separate channels each working independently and all in parallel. Figure 8 shows the MAC architecture suitable for any arbitrary moduli set. Block schematics of representative channels are shown in the Figures 9 (a), (b), and (c). Each prime field MAC channel has the design of Figure 9 (a). The \( i^\text{th} \) residues, \( x_i \) and \( y_i \) of channel \( j \) at the time interval \( t \) address two index ROMs that are used to find the logarithms \( z_i \) and \( z_2 \) corresponding to these operands. These indices are added using a modulo-\( n \) adder to get the index, \( x_0 \) of the product \( z_0. \)
Subsequently, $z_i$ addresses an inverse-index ROM to get back the product $z_i$, which then is transferred into Register1. The modulo adder in the next stage adds the present sum to the previous sum fed back from Register2, which is initialized to zero, thus accumulating the summation of the products $x_i y_i$ over the interval $i=1,N$. The final sum is left in Register2.

```
| X|m1 | Y|m1 | ΣXY|m1 |
|---|---|---|---|
| Channel 1 |
| ΣXY|m2 |
| Channel 2 |
| ΣXY|m3 |
| Channel n |
```

**Figure 8: MAC Architecture for an Arbitrary Moduli Set**

The integer-ring MAC channels are also designed on similar lines. The difference as apparent from Figures 9(b) and 9(c) is in the index adder section: since the integer mapping in $Z_{m^2}$ is done using an index triplet, three adders are needed in this case. For adding the $\beta$s, a modulo $2^{m-2}$ adder is required. In the case of $\gamma$, only an XOR gate is needed. To add the $\alpha$s, a conventional $m$-bit binary adder is sufficient. Similarly, in the case of $Z_{p^m}$, for odd $p$, to carry out the addition of the $\alpha$s and $\beta$s respectively, a $\varphi(p^m)$ modulo adder and a conventional $m$-bit binary adder are required. Since modulo adders will take up a major proportion of the MAC hardware, the performance depends on the modulo adders employed in the architecture.

```
| X|m2 | Y|m2 | ΣXY|m2 |
|---|---|---|---|
| Channel 2 |
| ΣXY|m3 |
| Channel n |
```

**Figure 9: MAC channels for (a) GF(p), (b) Z_{2m} and (c) Z_{pm}**
3.3 SCALING AND CONVERSION ISSUES

Scaling is an implicit requirement in RNS based MAC implementation because RNS multiplication increases dynamic range requirements geometrically. This is because the product of two RNS integers belonging to $\mathbb{Z}_M$ is defined in $\mathbb{Z}_{M^2}$. This geometric increase in dynamic range can rapidly fill any practical dynamic range limit. Scaling then manages the potential overflow problem. Scaling reduces the dynamic range of RNS variables below a potential overflow value. However, since scaling is a special form of division, it presents a major implementation problem. It requires a method by which an integer $X$, having an RNS representation, can be divided by some pre-specified constant $V$ with a minimum of overhead.

The forward conversion from binary to RNS and reverse conversion from RNS to binary are overheads involved in using RNS based processing. Forward conversion involves relatively simple calculations. A possible implementation [Stouraitis] is shown in figure 10. The input data word can be divided into 4 to 5 bits groups with each group performing parallel table look-ups. The outputs of the look-up tables are added in a modulo $m_i$ adder tree to get the modulo $m_i$ value. This approach can be easily mapped to FPGAs by using 4 or 5 inputs LUTs functions present.

Reverse conversion is based on either the CRT or MRC and requires significant overhead both in terms of area and speed. A wealth of approaches can be found in the literature on this topic as well on the combined output scaling issue. Some proposed techniques combine the scaling and reverse conversion. In addition, some implementations merge the reverse conversion computations into the multiply part of MAC by pre-multiplication of filter coefficients stored in the LUTs by factors used in CRT conversion. A direct CRT conversion architecture is shown in figure 11. The negative point in this approach is the requirement of multiple modulo M adders.

![Figure 10: Binary to modulo-m, RNS conversion for a 16-bit input, using four-table partitioning](image)
However, these modulo adders can take advantage of the fast carry chains present in the FPGAs for a fast implementation. However, the area overhead remains large.

Figure 11: RNS to binary conversion using the CRT reconstruction using LUTs and a modulo adder tree.

4. COMBINATION OF DA AND RNS TECHNIQUES

When implementing DA based MAC on FPGAs, the performance will deteriorate with increasing the number of products and the precision. This is obvious from the fact that as the number products increases, more level of decomposition will be required to map the DALUTs to LUTs available in FPGAs. This will result in more levels of adder stages with carry rippling. Increase in precision will also require a larger carry propagation chain. FPGA device families, such as Altera FLEX10K [Altera] or XILINX Virtex [Xilinx], are organized in channels (typically 8-bits wide). Within these channels are found short delay propagation paths, and dedicated memory blocks, with programmable address and data spaces that are commonly used to synthesize small RAM and ROM functions. Performance rapidly suffers when carry bits and/or data have to propagate across a channel boundary. This is called the channel barrier problem [Ramirez et al.]. Two’s Complement-DA designs encounter the channel barrier problem whenever word widths exceed the channel width in bits. An alternative design paradigm is to implement a fusion of DA and the residue number system. The RNS advantage is gained by reducing arithmetic to a set of concurrent operations that reside in small word length non-communicating channels. This attribute makes the DA-RNS potentially attractive for implementing MAC with FPGAs.

In RNS-DA implementation for every RNS channel, the current result in the accumulator must be multiplied by 2 prior to accumulating with the output of a LUT. This would require a specific LUT for 2 mod mj multiplication, which would add unwanted complexity to the recursive path. However, using a scaled modular accumulator [Ramirez et al.] computing,

\[ |y(n)| m_j = |2y(n-1) + x(n)| m_j , \]

an RNS-DA solution can be realized, leading to an efficient implementation of inner product. This accumulator is designed according to the following selection rules:

\[ 2 |y(n-1)| m + |x(n)| m \quad \text{if} \quad 2 |y(n-1)| m + |x(n)| m < m_j \]
\[ |y(0)| m_j = \begin{cases} 2 |y(n - 1)| m_j + |x(n)| m_j - m_j & \text{if } m_j \leq 2 |y(n - 1)| m_j + |x(n)| m_j < 2m_j \\ 2 |y(n - 1)| m_j + |x(n)| m_j - 2m_j & \text{if } 2m_j \leq 2 |y(n - 1)| m_j + |x(n)| m_j < 3m_j \end{cases} \]

A design of a modulo \( m_j \) scaling accumulator, for RNS-DA applications, is presented in [Ramirez et al.] and shown in figure 12. The accumulator uses one CPA (carry propagate adder) to realize the term \( 2 |y(n - 1)| m_j + |x(n)| m_j \), and two carry save adders (CSAs) used to compute \( 2 |y(n - 1)| m_j + |x(n)| m_j - m_j \) and \( 2 |y(n - 1)| m_j + |x(n)| m_j - 2m_j \) respectively.

These terms are computed concurrently and, as a result, have only one carry propagation stage in the critical path. The final result is selected on the basis of the carries generated in the summation stages as reported in Table 2.

Table 2: RNS-DA Scaling Accumulator decision logic table

<table>
<thead>
<tr>
<th>C0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>s1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>s1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>Any other combination</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

17
5. DESCRIPTION OF PROPOSED IMPLEMENTATION

The purpose of the undertaken implementation is to demonstrate the efficiency of DA and RNS based techniques in implementing multiply accumulate on FPGA architecture. For this purpose, some comparison metrics are essential as well as a uniform approach for a fair comparison. When referring to the fact that these techniques are efficient, the comparison is being made to other possible implementations of multiply accumulate on FPGA. A number of implementations of multiplier are available in the literature. Furthermore, constant coefficient multiplication is even a special case of multiplication. The DA technique is based on the optimization of constant coefficient multiplications and accumulations. The constants are the filter coefficients or any other values depending upon the algorithm. The common feature is that constant values do not change on per cycle basis but may change with a different filtering requirement (or in adaptive filters). In case of DA, this would require changing the contents of the ROMs to support a different set of constants. On the other hand, any suitable multiplication technique can be used with RNS; the small sized residues make possible LUT based multiplications with manageable sized ROMs. For sake of a fair comparison, a multiplication technique using constant coefficient multipliers will be used.

5.1 COMPARISON METHODOLOGY

A simple algorithm that uses a multiply accumulate as its core function is the Finite Impulse Response (FIR) filter. The functionality of an N-tap FIR filter is given by equation 8:

$$y[n] = \sum_{k=0}^{N-1} h[n] \cdot x[n-k]$$

Equation 8: A n FIR Filter Function

In equation 8, $y[n]$ is the output of the filter, $x[n-k]$ are the filter inputs and $h[n]$ are filter coefficients or constants. The $\cdot$ denotes the multiplication. The parameters that can be varied include the number of taps (N), the precision of input bits (B) and the precision of the
coefficients (C). Depending upon the values of these parameters, the precision of output y[n]
can be determined.

The implementations will involve varying the parameters N and B from 4, 8, 12 to 16 and 8,
16, 24 to 32 respectively1. Other variations will involve pipelining vs. non-pipelined
implementations. All these variations will be tested with three types of architectures, namely, DA
based, RNS based and DA-RNS based. Comparison will be made between area and speed
metrics of each corresponding implementation. A fourth implementation will use a non-DA and
non-RNS approach for multiply accumulate implementation. Two representative FPGA vendors
devices, namely Xilinx and Altera, will be targeted for implementation. The implementation
reports obtained from the vendor synthesis tools will be analyzed for performance metrics. The
structural implementations and the critical path reported by the tool will be compared to the
expected outcome by matching it with the device structure and the timing data detailed in the
datasheets.

5.2 DIFFERENCES FROM EARLIER IMPLEMENTATION PROPOSAL

In the first short draft proposal submitted for the project implementation, it was proposed
that a discrete cosine transform (DCT) algorithm will be chosen for implementation using DA,
RNS and possibly DA-RNS and DA-CORDIC techniques. This original proposal has been
changed after a detailed study carried out since draft proposal submission because there are too
many variations between architectural descriptions for DCT implemented using DA, RNS and
CORDIC techniques. These variations take complicate a fair comparison based solely on the
multiply accumulate technique being used. If a single architecture is used for all the different
techniques, it will result in obviously sub-optimal designs for techniques for which the
architecture was not optimized for, while, if the most optimized architecture is used for each
technique, it will not be easy to analyze whether the gains are a result of the novel design or the
technique itself. For these reasons, a simple algorithm such as a FIR filter function has been
chosen as it is free from complexities that were encountered in DCT implementation. This is
because a FIR filter is simply a multiply accumulate operation whereas in DCT algorithm other
steps are involved as well. Further, since CORDIC based implementations of generic filters are
superior to a multiply accumulate architecture based implementations [Yu92] and so the
CORDIC based architectures are not being considered for the implementation phase.

CONCLUSION

In this report, the necessary background for the implementation of DA, RNS and DA-RNS
based implementations of a vector dot product has been presented. Details of the proposed
implementation that will follow as well the methodology to be followed for comparison among
the different implementation have been discussed as well. This work will be followed by the
actual implementation results and their analysis for seeking the expected results presented in this
report.

BIBLIOGRAPHY


1 The number of parameter variations will depend upon time constraints for the implementations. At least two
combinations of each will be tested.


