Design and Evaluation of Random Linear Network Coding Accelerators on FPGAs

SUNWOO KIM, Hyundai Motor Company
WON SEOB JEONG, Yonsei University
WON W. RO, Yonsei University
JEAN-LUC GAUDIOT, University of California, Irvine

Network coding is a well-known technique used to enhance network throughput and reliability by applying special coding to data packets. One critical problem in practice, when using the random linear network coding technique, is the high computational overhead. More specifically, using this technique in embedded systems with low computational power might cause serious delays due to the complex Galois field operations and matrix handling. To this end, this paper proposes a high-performance decoding logic for random linear network coding using field-programmable gate-array (FPGA) technology. We expect that the inherent reconfigurability of FPGAs will provide sufficient performance as well as programmability to cope with changes in the specification of the coding. The main design motivation was to improve the decoding delay by dividing and parallelizing the entire decoding process. Fast arithmetic operations are achieved by the proposed parallelized GF ALUs, which allow calculations with all the elements of a single row of a matrix to be performed concurrently. To improve the flexibility in the utilization of the FPGA components, two different decoding methods have been designed and compared. The performance of the proposed idea is evaluated by comparing with the performance of the decoding process executed by general-purpose processors through an equivalent software algorithm. Overall, a maximum throughput of 65.98 Mbps is achieved with the proposed FPGA design on an XC5VLX110T Virtex 5 device. In addition, the proposed design provides speedups of up to 13.84 compared to an aggressively parallelized software decoding algorithm run on a quad-core AMD processor. Moreover, the design affords 12 times higher power efficiency, in terms throughput per watt than an ARM Coretex-A9 processor.

Categories and Subject Descriptors: B.2.1 **[Arithmetic and Logic Structures]:** Design Styles — *Parallel***;** B.5.1 **[Register-Transfer-Level Implementation]:** Design — *Styles***;** C.1.3 **[Processor Architectures]:** Other Architecture Styles — *Adaptable architectures*

General Terms: Design, Experimentation

Additional Key Words and Phrases: FPGA, Network Coding, High-Performance Decoder, FPGA Implementation, Galois Field Arithmetic, Parallel Architecture.

ACM Reference Format:
Kim, S., Jeong, W. S., Ro, W. W. and Gaudiot, J.-L. 2012 Design and Evaluation of Random Linear Network Coding Accelerators on FPGAs. ACM Trans. Embed. Comput. Syst.

DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This research was supported in part by the US National Science Foundation under Grant CCF-1065147 and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0013202). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US National Science Foundation or of the National Research Foundation of Korea.

Authors' addresses: S. Kim, Hyundai Motor Company, Republic of Korea; email: kayennez@gmail.com; W. S. Jeong, School of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea; email: wsjeong0528@gmail.com; W. W. Ro, School of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea; email: wro@yonsei.ac.kr; J.-L. Gaudiot, Department of Electrical Engineering and Computer Science, University of California, Irvine, USA; email: gaudiot@uci.edu.

DOI10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern data communication and computer network systems must provide enough bandwidth to handle an ever-increasing volume of data. In order to address this requirement, several techniques for maximizing network utilization have been introduced. Network coding is one of those techniques which can improve the data throughput of those networks with limited bandwidth [Ahlswede et al. 2000]. With this technique, network delay can be significantly reduced by utilizing the extra bandwidth capacity [Ho et al. 2006]. In fact, it can be implemented with various coding methods such as random linear coding [Chou et al. 2003; Ho et al. 2006].

One of the main problems when using network coding in practice is the high demand placed upon arithmetic resources; the encoding and decoding processes require extremely high numbers of Galois field operations as well as complex matrix computations, particularly when the size of the coefficient matrix and the amount of data to send are increased. This problem makes it difficult to practically exploit the advantage of network coding [Shojania and Li 2007].

Past work on acceleration of network coding has been mostly based on new algorithm development [Li et al. 2003; Koetter and Médard 2003] or on parallelization technique expansion for general-purpose processors [Shojania and Li 2007; Park et al. 2009; Park et al. 2010; Kim et al. 2012] and heterogeneous multi-core processors [Kim et al. 2011]. In addition, several recent projects have introduced parallelized network coding on graphics processing units [Shojania et al. 2009a; Shojania and Li 2009b; Lee and Ro 2010]. However, these approaches have not considered the problems associated with low-power embedded systems or portable devices and may not be suitable for devices other than those based on high-performance general-purpose CPUs. Using high-end multi-core microprocessors or graphics processing units (GPUs) can cause serious problems in most small network devices, since those devices are expected to have low power consumption. Such devices range from portable handheld computers and smart phones to popular network devices such as access points and repeaters.

To enable network coding on intermediate nodes, we could probably use network processors, such as the Cavium Octeon series, or embedded processors based on ARM. However, we would not have enough computational power with these processors, which consequently may cause low performance with long decoding delays. As a compromise between high-end general-purpose processors and low-end embedded processors, we here propose using a separate piece of logic on an FPGA chip for the decoding process.

One could dispute the necessity of decoding for access points or repeaters, since the decoding process is often only required by the end-users. However, there are often also reasons for the intermediate nodes to decode the data. In fact, if the intermediate node directly connected to the destination node can perform some pre decoding, the overhead of the decoding process can be hidden from the user in the destination node. Otherwise, the decoding may place excessive computational demands upon the destination node, which would negatively affect the end-users.

Interestingly, network coding is not a fixed protocol. For instance, the size of a block and the number of blocks may change depending on the network circumstances. With a general-purpose processor, such changes in block sizes are not a significant problem since an algorithm written in a high-level language is relatively easy to modify. However, when developing a decoder with customized hardware logic, changes in block sizes require a certain level of reconfigurability. This means that the inherent reconfigurability of FPGAs will be advantageous when coping with changes in the specification of network coding.

In this paper, therefore, we introduce a reconfigurable decoding logic implemented in FPGA technology. This logic is meant to function as a high-performance network

coding accelerator in embedded systems. We designed it to manage the computational overhead of arithmetic operations and enhance the decoding speed. Our main idea is the improvement of the decoding speed by dividing and parallelizing the entire decoding process. The proposed design mainly focuses on matrix manipulation and Galois field operations in order to expedite the decoding procedure. In fact, the field programmable gate array (FPGA) technology is widely used in various fields such as network systems [Mehrotra et al. 2004] and cryptography [Chelton and Benaissa 2008; Mace et al. 2008].

The main contribution of this paper is to show the practical designs and their trade-offs in implementing network coding decoders with FPGAs. For this purpose, the parallelized design for Galois Field operations and Gauss-Jordan elimination is proposed in considering practical network coding applications. To the best of our knowledge, this is the first research project proposing a hardware accelerator which can be practically used for network coding on FPGAs. One distinct feature of the proposed design is that coefficient vectors as high as 256 and 512 can be handled with this design. In addition, the power saving in using the proposed design in embedded systems is verified.

The performance of the proposed idea has been evaluated and compared to that of the decoding process of a network coding software algorithm executed by an Intel quad-core processor working at a frequency of 2.66 GHz and an AMD quad-core processor at a frequency of 2.20 GHz. Moreover, we compared with the performance and power efficiency of the decoding operation on a real ARM processor, the Cortex-A9. The maximum throughput of 65.98 Mbps is achieved with the proposed FPGA design using an XC5VLX110T Virtex 5 device. In addition, the proposed design provides maximum speedups of 13.84 and 6.73 compared to the aggressively parallelized software decoding algorithm run on quad-core AMD and Intel processors, respectively. Furthermore, the design shows maximum 12 times higher power efficiency than an ARM processor.

The rest of this paper is organized as follows: Section 2 covers background research on network coding and Galois field arithmetic operations. In Section 3, the decoder designs for the two different methods including the description of the decoding process flow is given. The Galois Field ALU (GF ALU), specially designed for the Galois Field operations, and its parallelized architecture are also described. At the end of this section, the scalability issue and advantages of reconfigurable hardware network coding accelerator are covered. In Section 4, our simulation methodology, the simulation and experimental results are presented along with the performance analysis and power efficiency. Section 5 concludes this paper.

2. BACKGROUND RESEARCH

Network coding is a data coding technique that has been introduced to enhance the throughput and reliability of data networks.

2.1 Concept of Network Coding

To illustrate network coding, Fig. 1 uses a simple butterfly model (which was first used by Ahlswede [2000]). Let us assume that the source node, S, is attempting to multicast two single data bits, b_1 and b_2, to two destination nodes, Y and Z, through the networks. Both networks in Fig. 1 have the same nodes (T, U, W, and X) and links, a single source node S, four intermediate nodes, and two destination nodes Y and Z. We assume that each link between the nodes has equal bandwidth and that a single data bit can be transmitted per unit time.

Fig. 1(a) represents an example of a traditional network system without network coding. Let us assume that the two data bits arrive at the intermediate node W at the same time. Since the link can only forward one of the bits, not both, the node W

should forward one data bit and store the other. Meanwhile, each destination node has already received one data bit and is awaiting the other. However, only one bit can be transferred to node *X* and, depending on the data bit selected at the congested node, one destination node should wait for another time unit to receive both data bits. In this network model, the link between the nodes *W* and *X* becomes a bottleneck.

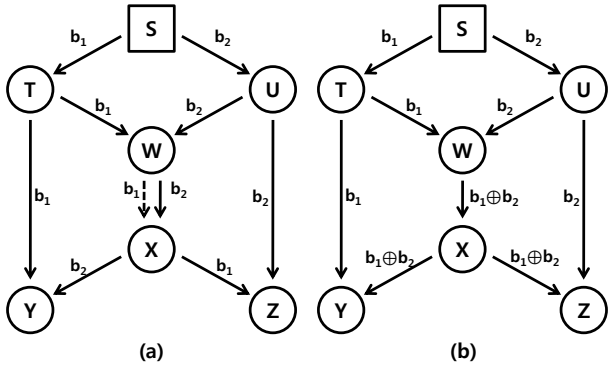


Fig. 1. (a) Simple acyclic network. (b) Network with XOR.

In contrast to the first example, we can apply a simple XOR operation of the two data bits in the network as shown in Fig. 1(b). If we ignore the overhead of performing the XOR, the congested node *W* can send the result of $b_1 \oplus b_2$ in a single unit of time. Eventually, the destination nodes, *Y* and *Z*, receive (b_1 and $b_1 \oplus b_2$) and (b_2 and $b_1 \oplus b_2$), respectively. By applying another XOR operation to the data received, each destination node can retrieve the original data bits. In this example of network coding technique, the networks can have higher bandwidth utilization, especially when multicasting for data communications is used. With an increase in bandwidth utilization, network coding can efficiently reduce data network delay. Each node in the network can find an optimal path to transfer data [Chou and Wu 2007]. In addition, the probability of a successful transfer and the robustness of network also can be improved, which consequently helps to reduce data transfer delays [Ho et al. 2006].

In practice, network coding is not as simple as an XOR. In fact, there exist several coding methods within the category. In this study, we have chosen random linear coding [Chou et al. 2003; Ho et al. 2006] since this is known to be asymptotically optimal in any network system. In random linear network coding, the elements of the coefficient matrix are selected at random from the finite filed and one first packetizes the data to produce encoded data frames [Chou and Wu 2007]. Then, the data frames are encoded with a group of coefficient vectors through the matrix multiplication operation.

When we multiply two numbers in any binary number system, the result normally requires more bits than the operands. On the other hand, any multiplication with two Galois field numbers is closed within the Galois field. In other words, by considering the data elements as the numbers in a set of a Galois field, all the computational results can be represented with a fixed number of bits. The size of the field varies with the application, but we use GF(2^8) for the size of the Galois field in this paper. As mentioned previously, the size of the Galois Field used in our design for network coding applications is as small as GF(2^8) so that the complexity as well as the computation delay can be correspondingly reduced since all numbers can be expressed in 1 byte.

To encode the original data, we need a group of coefficient vectors, which forms the coefficient matrix. Then, the encoding operation can be represented as $e^T = C_n \cdot o^T$ where $C_n^T = [\, c_1^T \cdots c_n^T \,]$. In these equations, e^T is the encoded data vector, c_n^T is a

coefficient vector, and o^T is the original data vector. The superscript T denotes a transposition and each vector contains n elements. To produce an encoded data vector from an original data vector composed of n elements, the coefficient matrix must be of size $n \times n$. Each element that belongs to the coefficient matrix is randomly chosen from among the set of Galois field numbers.

Since the coefficient matrix traverses the network along with the encoded data, it is not efficient to use a coefficient matrix for encoding only a single data vector. Therefore, a data matrix is composed by combining several data vectors and generating an encoded data matrix such as $E_N = C_n \cdot O_N$ where $E_N^T = [\, e_1^T \cdots e_N^T \,]$, $C_n^T = [\, c_1^T \cdots c_n^T \,]$, and $O_N^T = [\, o_1^T \cdots o_N^T \,]$. The coefficient matrix thus generates an encoded data matrix that is composed of N data vectors.

In order to retrieve the original data from the encoded message at the receiving node, the source node sends the coefficient matrix together with the encoded data matrix. Hence, the coefficient matrix and the encoded matrix are combined as follows: $C_n^T \mid E_N^T = [\, c_1^T \cdots c_n^T \mid e_1^T \cdots e_N^T \,]$. A coefficient matrix C_n of size $n \times n$ and an encoded matrix E_N of size $n \times N$ are concatenated to form a matrix of size $n \times (n + N)$. The combined matrix is divided into row vectors and packetized when transmitted. Each row vector is recognized as a block and thus a total of n blocks are generated from a concatenated matrix. The blocks will have a packet identification number at the end and will compose a frame to be transmitted.

Once data is transmitted, a receiving node can decode the original data based on the coefficient vector attached to the beginning of each block. After receiving enough blocks, a receiver node has sufficient information to perform the decoding with the encoded matrix E_N and the coefficient matrix C_n. This implies that the original data matrix O_N can be retrieved by simply multiplying the inverse of the coefficient matrix, C_n^{-1} by the encoded matrix E_N. That is, the decoding operation can be accomplished by simply multiplying the inverse of the coefficient matrix or by directly applying the Gauss-Jordan elimination method to the encoded data. However, either way, at least n independent blocks must be received in order to obtain the inverse of the coefficient matrix.

2.2 Arithmetic Operations over Galois Fields

In Galois fields, addition and subtraction are equivalent; both can be performed by a bitwise XOR operation. However, multiplication and division are more difficult and more computationally expensive tasks. Generally, the execution of these operations is based on the extended Euclidean algorithm (EEA) or Fermat's little theorem [Deschamps and Sutter 2006]. Many architectures exist based on the EEA or Fermat's little theorem [Yan and Sarwate 2003; Kim and Hong 2002; Itoh and Tsujii 1988]; however, these are usually inefficient in terms of hardware complexity and execution time.

Those previously proposed designs require a complex computational logic unit with a larger Galois field size than in our design. As mentioned above, the size of the Galois field used in our design for network coding applications is as small as GF(2^8) so that the complexity as well as the computational delay can be correspondingly reduced. In this study, we have experimented with two possible types of GF ALUs: the table lookup method and the computation oriented method.

The first proposed method consists in using a pre-calculated table to obtain the results of Galois field operations. By its nature, this approach is algorithm-independent since it is a simple table lookup: the logarithm and antilogarithm tables provide a deterministic and reduced execution delay. Multiplying two numbers is simply an addition when we take the logarithms of those numbers. Similarly, a division operation is a subtraction. This observation implies that we can substitute addition and subtraction for multiplication and division if we take the antilogarithms

of the logarithms of Galois field numbers. Since we use Galois field numbers, the size of the tables is bounded by the size of the set.

The second possible approach is to use a combinatorial circuit for multiplication operations. As we have a relatively small set of numbers, the logic for multiplications can be implemented by our target FPGAs. For the division operations, the extra logic to calculate the reciprocals is placed in front of the multiplication logic. The performance analysis of both designs will be given in Section 3.

2.3 Related Work

The fundamental concept of network coding was introduced by Yeung and Zhang [1999] for satellite communication networks. It was subsequently more fully developed by Ahlswede et al. [2000].

In fact, there exist various coding methods. Pedersen et al. [2008] implemented and evaluated XOR-only network coding. Chachulski et al. [2007] and Katti et al. [2008] proposed the use of random linear network coding with Galois field numbers. Several algorithms and architectures such as LFSR multipliers [Lin and Costello Jr. 1983] and Mastrovito multipliers [1991] have been proposed for arithmetic operations with Galois field numbers. However, a complex associated hardware unit is needed.

Shojania and Li [2007] first introduced the concept of parallelized progressive network coding (PPNC). In network systems, the data frames are transferred one by one. The arrival of frames is not continuous and the time gap between each frame may randomly change depending upon the network conditions. Therefore, in network coding, it is not optimal to wait for entire data frames to arrive in order to form a matrix. The main idea of PPNC is to perform some partial decoding upon reception of each data frame.

Although PPNC has suggested a parallelized network coding method, the algorithm fundamentally cannot fully exploit the possible parallelism since the workload in each parallel task is unbalanced. For more balanced workloads, Park et al. [2010] have developed three different methods, of which dynamic vertical partitioning (DVP) yields the best performance. This algorithm is mainly based on the idea of assigning column-wise workloads to the threads. Since the number of columns remaining to be decoded changes during the decoding process, the algorithm dynamically assigns a balanced number of columns to each thread so that the workload can be evenly distributed.

Shojania et al. [2009a; 2009b] and Lee and Ro [2010] have implemented GPU-based network coding. The performance of parallelized network coding with GPUs was compared to that with CPUs in the Shoania's research. Lee and Ro proposed DSD, GPU-based parallel progressive decoding algorithm. These researches have shown promising results; however, a GPU inherently requires the support of a CPU and cannot be efficiently utilized in a small embedded system. In recent research, network coding has been implemented on hand-held devices [Shojania and Li 2009c]. The iPhone and iPod Touch were chosen as the devices, and the evaluation of the ARM processors used in those devices was also performed.

Previously, Gulati and Khatri [2008] discussed adapting network coding to FPGA technology. The main contribution of their work was to improve the internal routing capability of FPGAs by using the network coding technology. In addition, Nagarajan et al. [2010] proposed a possible design for Galois field hardware architecture and analyzed the possible hardware implementation. Their approach is purely based on an application-specific integrated circuit that is not reconfigurable and is not sufficiently flexible to cope with possible changes in network coding specifications.

Although the primitive approach of using FPGAs for a hardware accelerator for network coding has been published previously [Kim and Ro 2010; Kim and Ro 2012], to the best of our knowledge, this is the first research work that proposes a hardware accelerator which can be practically used for network coding on FPGAs. One distinct feature of the proposed design is that a coefficient vector as large as 256 or even 512 can be handled with this design. In fact, Yoon and Park [2010] implemented FPGA-based acceleration engines of Gaussian eliminations on finite Galois field. They implemented parallelized decoding algorithm, however, the block size was fixed and too small to be practically used. In addition, they did not discuss about the advantages of reconfigurability.

One other distinct feature of our research is that the power saving which the proposed design yields for embedded systems is verified. In fact, we show the power consumption of a real ARM processor in the decoding process and compared the power efficiency of our design with the power efficiency of an ARM processor.

3. HIGH-PERFORMANCE NETWORK CODING DECODER IMPLEMENTATION

In this section, the proposed network coding decoder designs and various FPGA implementations are described in detail. In addition, we discuss some implementation issues related to scalability.

3.1 Overview of the Proposed Architectures

To provide a better understanding of the proposed decoder architecture, the overall block diagrams of the proposed designs are shown in Fig. 2. Indeed, we have proposed two possible design choices by considering the implementation trade-offs and the FPGA characteristics:

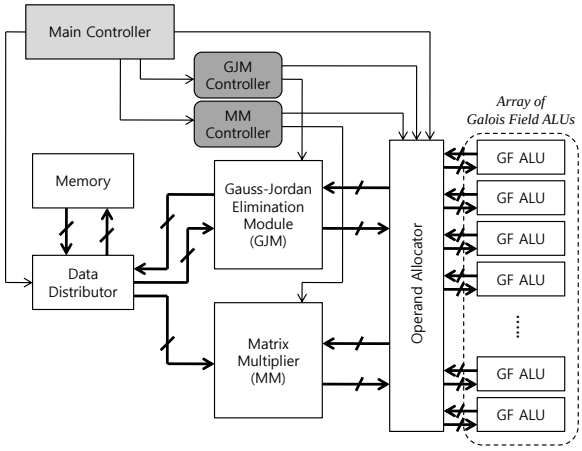
- Inversion first (INVF). Finding the inverse of the coefficient matrix using Gauss-Jordan elimination and then multiplying the encoded data matrix by the inverse of the coefficient matrix to obtain the original data matrix.
- Full Gauss-Jordan (FGJ). Performing the Gauss-Jordan elimination on the entire dataset (the coefficient matrix and the encoded data matrix) to directly obtain the original data matrix.

The decoder module is composed of several sub-modules as described in Fig. 2. In these designs, the main controller provides the control for the sub-modules. In the case of INVF, there are two main functional modules: the Gauss-Jordan elimination module (GJM) and the multiplier module (MM). These modules are under the control of two small controllers: the GJM controller and the MM controller, which work as dedicated controllers to provide more precise control of each module. In the case of FGJ, however, the design is simpler, since there is no need for an additional module for the matrix multiplication. The rest of the sub-modules, including the memory module and the GF ALU array, are identical in both approaches.

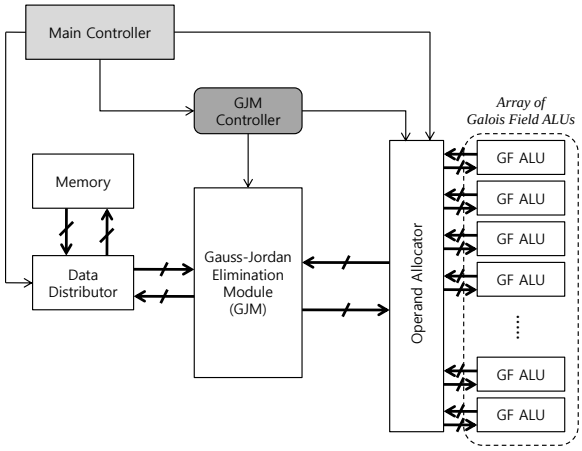
3.1.1. Inversion First Method (INVF). The block diagram of the decoder module for the INVF method is shown in Fig. 2(a). The decoder is composed of functional sub-modules that are controlled by the main controller. At the beginning of the decoding process, the main controller allows the encoded data from outside the decoder to be stored in the memory through the data distributor. All data paths between the memory and the two functional modules (GJM and MM) are connected through the data distributor, which is also controlled by the main controller.

All the necessary computations are performed on the array of GF ALUs. Each GF ALU has the capability of computing two 8-bit Galois field numbers for addition, subtraction, multiplication, and division. The array is shared by the two functional modules, GJM and MM, through the operand allocator. In fact, the main controller

gives the control signal to the operand allocator and specifies which module to use for GF ALUs. The operand allocator is also in charge of distributing the bit stream of operands from the functional modules to each GF ALU.



(a) Inversion first (INVF) design



(b) Full Gauss-Jordan (FGJ) design

Fig. 2. Block diagrams of two proposed network coding accelerators.

The decoding process of the INVF implementation is divided into the three stages listed in Table I. In the first stage, the main controller collects the encoded data. As described above in Section 2, the incoming encoded data is in the form of separated frames. New data is only received when the decoder is not in the decoding process. If the decoder is idle (i.e., ready to accept new encoded data) the main controller collects data frames from the outside until a sufficient number of independent data frames are stored in the memory. The decoder module does not proceed to the next stage until it stores fully independent coefficient vectors. Therefore, there is a step to check that all the coefficient vectors are independent. After that, the accelerator proceeds to the next step by sending a trigger signal to the GJM.

Table I. Operation of Each Stage in Decoding Process

Stages	Task Descriptions
1	The decoder collects the encoded data and stores that in memory. All the coefficient vectors are checked for independence.
2	The inversion of the coefficient matrix is performed. This process is mostly controlled by the GJM controller.
3	The original data is retrieved by the matrix multiplication.

The GJM performs Gauss-Jordan elimination on the coefficient matrix and produces the inverted matrix in the second stage. The flowchart of the process is shown in Fig. 3. For Gauss-Jordan elimination, the GJM has two register files: *RC* to hold the coefficient matrix and *RI* to hold the inverted matrix. At the beginning of the stage, the module initializes *RI* as an identity matrix while the coefficient matrix is loaded from the memory and stored in *RC*. Once the initialization is finished, the GJM performs the matrix inversion.

When the two register files are ready, the GJM begins the Gauss-Jordan elimination process. The Gauss-Jordan elimination method that we apply is general and mostly consists in a routine for selecting pivots and performing arithmetic operations on each row. The selection of a pivot and the rows to calculate is made in the *Check Status* state and the corresponding calculations are performed in the *Row Operation* state. At the end of the process, the content of *RC* is a matrix where the diagonal elements are not normalized and all others are zero. Therefore, there is an operation for normalization at the end of the routine to complete the process. After that, the inverted coefficient matrix is written back to the memory.

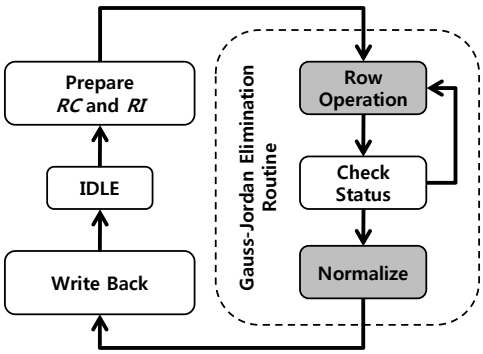


Fig. 3. Flowchart of Gauss-Jordan elimination module.

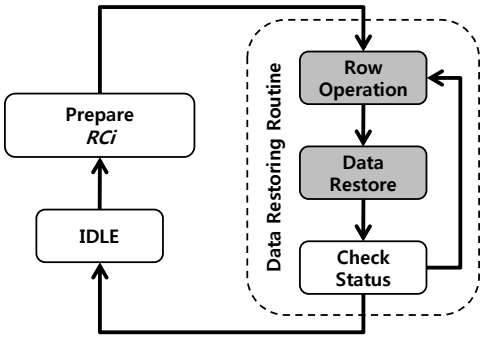


Fig. 4. Flowchart of matrix multiplier.

Finally, the MM performs its operation in the last stage of the decoding process as depicted in Fig. 4. Like the GJM, the MM has its own register file, which we call *RCi*, to contain the inverted coefficient matrix and thus the size of the register file is the same as that of the inverted coefficient matrix. Once a control signal from the main controller triggers the MM, the module loads the inverted coefficient matrix and the encoded data from the memory and holds these in the register files. Afterwards, the last step of the decoding process is executed; the last process consists of multiplying the encoded data matrix with the inverted matrix. The computational results are XORed and continuously added until the target data can finally be obtained.

During the second and third stages, the GJM and the MM can utilize the GF ALUs. These states are depicted as shaded boxes in Fig. 3 and Fig. 4. Most of the calculations are related to the matrix operations on the coefficient matrix and its corresponding data matrix. The number of GF ALUs matches the number of rows in the coefficient matrix for the small coefficient vector. This implies that simultaneous execution of all the required operations for any row or column is possible to achieve. More detailed descriptions of the GF ALUs and their parallel structure will be provided in the subsequent sections. The main difficulty in improving the performance of INVF is that in the third stage we must access the memory to load each column of encoded data recursively. Since the size of the matrix increases exponentially as the number of rows increases, the recursive access results in performance degradation as the coefficient matrix becomes larger.

To address the performance degradation due to the successive memory accesses, we implement a buffer block in the MM. The requirement of the buffer is that the row-wise data from the memory can be written into the buffer while the column-wise data in the buffer can be accessed at once. In order to accomplish this, we accumulate row-wise buffer strips to implement a buffer block. Each buffer strip is synthesized as a basic single-port embedded RAM that holds row-wise encoded data loaded from the memory and that can be accessed for each byte (which is the size of each data element). Once a column is requested by the MM controller, the buffer block decodes the requested address and reads each data element that belongs to the requested column. These data elements are concatenated to form the column data stream and forwarded to the operand allocator as operands for multiplication.

One more possible enhancement is to incorporate a pipelined execution model for the Gauss-Jordan elimination procedure and the matrix multiplication operation. In this paper, we have not considered such a model, in order to guarantee a fair comparison between INVF and FGJ. We leave the idea for future work.

3.1.2. Full Gauss-Jordan Method (FGJ). The FGJ method is based on the idea that the matrix inversion and the multiplication can be executed simultaneously. In the case of the INVF method, the purpose of using Gauss-Jordan elimination is to obtain the inversion of the coefficient matrix. If we replace the identity matrix with the coded data matrix, we can directly obtain the original data and the decoding process by applying a Gauss-Jordan elimination procedure on the entire dataset.

In this design, the GJM has two register files: *RC* to hold the coefficient matrix, as with the previous designs; and *RD* to buffer the encoded data. The reason that we separate the registers for the coefficient matrix and the data is that the coefficient matrix is more frequently accessed in order to configure the pivot and leading elements of the currently computed rows. Similar to the buffer block for the MM in Subsection 3.1.1, *RD* holds a part of the encoded data matrix. The main purpose of *RD* is to provide data to the ALU array.

In the proposed design, one of the important aspects to accelerate the decoding process is the massive parallelization of arithmetic units. This implies that we need to supply sufficient data to the GF ALU array to exploit the parallelism. Otherwise,

the arithmetic units must wait for their operands and this consequently introduces additional overhead.

For a precise comparison of the two decoding methods, other features of the decoder logic such as memory bus width, the size of GF ALU array, and the operand allocator are implemented identically. We expect there exist trade-offs between INVF and FGJ.

In fact, the FGJ needs one additional cycle to complete the row calculation as the column size of the data matrix becomes larger than the number of GF ALUs, whereas the INVF can complete each row or column in one cycle. However, when the number of GF ALUs becomes larger as the coefficient matrix expands, the number of steps required for the row calculation is reduced. Likewise, the data access pattern affects the overall performance. The FGJ method is able to load the coefficient matrix and the encoded data from the memory to the buffers at one time, but the INVF method needs to load and store an inverted matrix in additional cycles.

When processing the multiplication stage in the INVF method, the data access is necessarily column-wise since a row of the inverted coefficient matrix and a column of the encoded data matrix are processed together. Conversely, the FGJ method does not require any column-wise access as the multiplication stage is removed and the data transfer can be done in a single transaction.

From an architectural viewpoint, there are several advantages to using the FGJ method rather than the INVF method. First, we do not need any sub-modules for the matrix multiplication and its attached control logic, as shown in Fig. 2(b). This can eventually provide area efficiency as we will show later in this paper. Second, there is no recursive access to the encoded data in the memory for the matrix multiplication.

As described earlier, the performance degradation is mainly due to the increased number of memory accesses caused by the recursive computations. In addition, there is no burden of accessing data in the column-wise direction since the matrix multiplication is not performed.

3.2 Scalability to Large Coefficient Vector

So far, there has been no standardization or agreement on which protocol to use in practical network coding. This implies that scalability is an important issue for decoder implementation. The decoder should be ready to handle any possible change in size. In this subsection, we present our design of additional features to maintain the performance despite the large size of the coefficient matrix. The major modifications are as follows:

- Implementing an embedded memory that supports a 1024-bit bus width inside an FPGA device.
- Introducing microcode for the main controller instead of using a finite state machine.

The main bottleneck of the decoder designs in the previous subsections is caused by the limited memory bus width. As mentioned, capability to provide a sufficient number of data elements is an important factor to improve the overall performance. For that purpose, we implement an embedded memory with a large data bus width. We believe that this can provide a sufficient number of operands directly from the memory. In our design, the proposed embedded memory is implemented using a Xilinx LogiCORE IP Block Memory Generator [Xilinx 2009a]. The generator supports a maximum of 1152 bits of data width, which is why we selected 1024 as the bus width. As a result, it is possible to transfer all the required data elements from the memory in a single transaction.

For this extended design, the microcode is offered as a substitute for a finite state machine in order to simplify the design of the controllers. As the size of the coefficient matrix becomes larger than the number of GF ALUs in the array, simultaneous computation for a single row becomes impossible to finish in a single cycle. Therefore, the procedure for a single row and a single column should be performed over multiple cycles. Moreover, by using the embedded memory as a separate module, the main controller has to handle more complex control states. The newly designed decoder does not require any RTL modification to support both methods. This can be accomplished by modifying the microcode inside the main controller.

With the modified decoder design, we have successfully implemented the decoder using both INVF and FGJ methods for sizes of 256 and 512. Since the data bus allows row-wise accessing of the data, the implementation is suitable for the FGJ method. The INVF method includes the overhead of column-wise access. This problem could be handled by storing the data in column-major order in the embedded memory; however, we intend to concentrate on the FGJ method based on the results of the design study for smaller size. As described in the previous subsections, the FGJ method yields the best results from the viewpoints of both the performance and the area overhead as the size of the coefficient matrix becomes larger.

3.3 Galois Field Arithmetic Logic Unit

As described in the previous subsection, we have implemented both types of GF ALU to support the addition, multiplication, and division operations over a Galois field. The corresponding structures are depicted in Fig. 5.

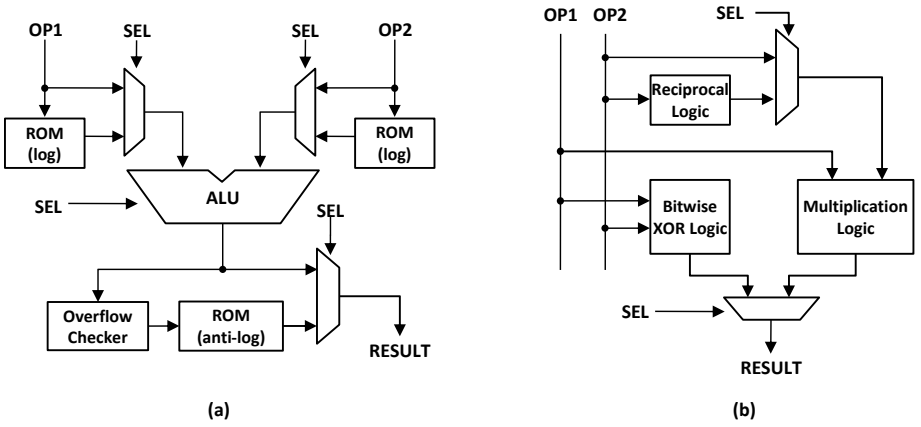


Fig. 5. Structures of GF ALUs using (a) lookup tables and (b) computational logic.

3.3.1 GF ALU using Lookup Tables. In this structure, the ALU consists of two ROMs each containing a logarithm table, one more ROM containing an antilogarithm table, and an overflow checker. The Galois field numbers in 8-bit representation are converted to their corresponding logarithms by looking up the values in the two logarithm tables. Similarly, the antilogarithm table is used for the opposite conversion. The overflow checker determines whether an overflow occurs during the current computation.

The signals input to the GF ALU logic are two 8-bit operands from the operand allocator. The ALU performs arithmetic operations, including multiplication, division, and addition or subtraction, on these two operands according to the select signal. In the case of an addition or subtraction, the operation is relatively simple:

there is no need to use lookup tables, since addition and subtraction of Galois field numbers are identical and can be performed by a bitwise XOR operation.

In the case of multiplication and division, however, the calculation cannot be executed by a simple bitwise operation. According to the select signal, the outputs of the two ROMs are chosen as the two operands of the ALU if the operation is a multiplication or a division. These outputs are the converted logarithms of the original input operands. By taking the logarithms, multiplication and division are replaced by addition and subtraction, which are much simpler. The arithmetic result of the logarithms is then converted back to its original value by searching the antilogarithm table after passing through an overflow checker.

Although multiplication and division are replaced with addition and subtraction by taking the logarithm, the overflow of addition and subtraction operations must still be dealt with. This can be simply represented by modulo operations of the largest value in the set, which is $2^8 - 1$ in the case of GF(2^8). For that purpose, we implement an overflow check module. As soon as the computation in the ALU is completed, the result is sent to the overflow checker and is examined to determine whether there has been an overflow during the computation. The output of the overflow checker is then reconverted by looking at another ROM containing an antilogarithm table. An additional MUX chooses the final result of the GF ALU between the raw result of the ALU and the converted result.

3.3.2 GF ALU using Computational Logic. Taking the logarithm of a Galois field number can reduce the complexity of a computation, but it causes additional overhead in accessing the ROMs. Therefore, we replace the table lookup method with an implementation of the Galois field computational logic in this part.

As with the table lookup method, addition and subtraction operations are not major performance issues. In order to expedite the multiplication operation, we have implemented logic to multiply two 8-bit Galois field numbers. The logic is originally from the algorithm of loop-based multiplications in GF(2^8), which is shown in Fig. 6. In the original code, each bit is processed and shifted by one bit for an iteration of the loop. In our implementation, instead of loop iteration all bits are simultaneously processed through the logic for a faster computation.

For division operations, we implement reciprocal logic that provides the multiplicative inverse of a single 8-bit Galois field number. The logic is obtained from eight Boolean equations of the truth table, which maps the reciprocal of each entry in the set. In order to reduce those equations, we have applied the Quine-McCluskey algorithm by using qmc (version 0.94), a Boolean logic simplifier that is provided as a Debian software package [Debian].

```
byte gfmult(byte a, byte b) {
    byte p := 0;
    byte cnt;
    byte HbitSet;
    for(cnt=0; cnt<8; cnt++) {
        if((b & 1) == 1)
            p := p ^ a;
        HbitSet := (a & 0x80);
        a := a << 1;
        If(HbisSet == 0x80);
            a := a ^ 0x1b;
        b := b >> 1;
    }
    return p;
}
```

Fig. 6. Pseudo code of loop-based multiplication in GF(2^8).

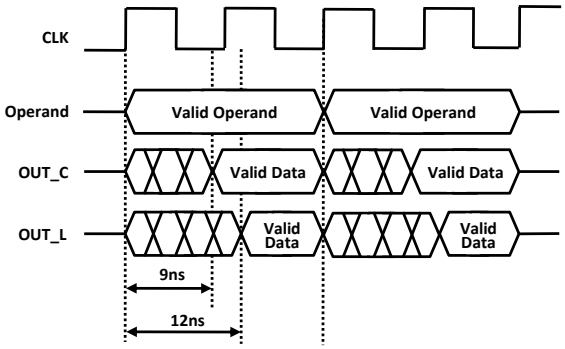


Fig. 7. Timing diagram of two GF ALUs for multiplication at frequency of 100 MHz.

The performances of the two types of GF ALU are evaluated on the target device. As shown in the timing diagram in Fig. 7, a multiplication by the GF ALU using computational logic affords better performance. OUT_C and OUT_L in the timing diagram represent the valid output of GF ALUs using computational logic and lookup tables, respectively. The propagation delay of multiplication is adequately reduced by simplifying the combinational logic for calculation. As shown in Fig. 5, a multiplication uses only one combinational logic part and makes the delay smaller than for accessing ROMs. Although multiplication using a computational logic GF ALU is faster than that using a lookup table GF ALU, the respective delays in addition, subtraction, and division are almost the same in the worst case.

The GF ALUs using computation logic have better performance; however, the area usage of computation logic is worse than lookup table. In fact, computation logic uses more LUTs than look up table. Regarding this, we will show the trade-off between performance and area overhead in Section 4. Each GF ALU can be chosen by any decoding method, INVF and FGJ. Moreover, flexibility to choose decoding method based on FPGA's reconfigurability, can give optimized solution to system depending on a required decoder spec and design constraints in various network coding circumstances.

3.4 Operand Distribution

As described briefly in the previous subsections, the number of GF ALUs is equal to the size of the coefficient matrix in the case of small coefficient vectors. These ALUs are connected to the operand allocator through the two bit distributors. Each GF ALU operates independently from the other ALUs and thus can execute 16, 32, 64, or 128 GF arithmetic operations in parallel, depending on the size of the coefficient matrix. In the case of larger implementations, the number of GF ALUs remains 128, considering the area overhead. Then, each row of the data is divided into blocks of 128 elements and computed sequentially.

Fig. 8 depicts the structure of the operand allocator and the GF ALUs. The operand allocator accepts two operands each as inputs from the GJM and the MM in addition to a control signal from the main controller. The control signal contains the select signals, which convey the input to be used and the expected operation. Each input operand from the functional modules is a sequence of 8-bit data elements that consists of a row from either the coefficient matrix or the encoded data matrix. The chosen operand sequences are then sent to each bit distributor, which parse the input into 8-bit operands and distribute those to the proper GF ALUs. While the operands are distributed through the bit distributors, the control signal for the GF ALUs is distributed in the same manner.

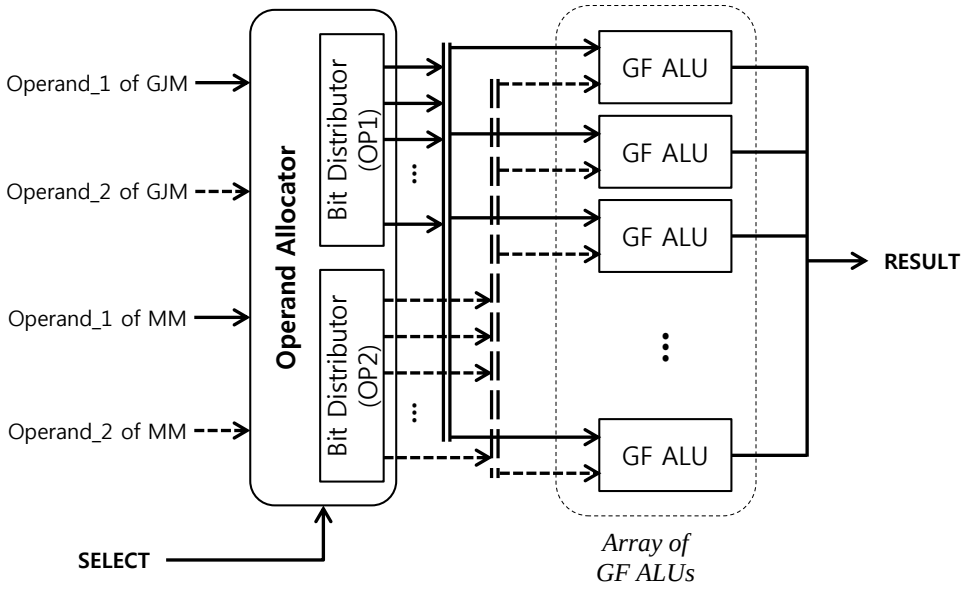


Fig. 8. Connection of ALU control module and GF ALUs.

Obviously, information to select the input for the control signal is not necessary when using the FGJ method. The decoder has a sole functional module and the ALU array is fully dedicated. Fig. 8 depicts the structure in the case of the INVF method. However, since all the processes other than that to select the input operand sequences are identical to those used in the FGJ method, the figure and its description of the operand distribution can be equally applied to the decoder using the FGJ method.

3.5 Advantages of Reconfigurable Network Coding Accelerator

There is no standard fixed protocol for network coding applications. For instance, the size of a block and the number of blocks may change according to the target network systems and policy. Thus, to implement a more efficient decoding process, network coding accelerators should change its structure accordingly with a certain level of reconfigurability. This means that the inherent reconfigurability of FPGAs would be extremely useful especially when it comes to maneuvering changes in the specification of network coding.

Although we can implement parallelized accelerators in ASIC and give a certain level of flexibility with firmware control, the manufacturing cost and the design overhead including time to market become higher than FPGA. Moreover, based on our simulation results, changing the decoding methods and the structures of GF ALUs makes FPGA solutions more attractive than using a fixed structure.

Software decoding algorithms on DSPs can be a solution as well to solve the flexibility problem; however, the decoding process can cause a huge overhead in a processor and degrade the overall performance. From the above observations, implementing network coding accelerator in FPGA provides more practical advantages than other solutions.

4. PERFORMANCE EVALUATION

In this section, we present the methodology of our performance evaluation. Subsequently, the synthesis results of our implementation on the FPGA device are shown and the performance results are presented.

Table II. Desktop Systems for Software Algorithm

	System A AMD Quad-Core	System B Intel Quad-Core
CPU	AMD Phenom-X4 9550	Intel Core 2 Quad Q9400
CPU Clock	2.20 GHz	2.66 GHz
RAM	4 GBytes	2 GBytes
Cache Size	L1: 4 × 128 KBytes L2: 4 × 512 KBytes L3: 2 MBytes Shared	L1: 4 × 64 KBytes L2: 2 × 3 MBytes
OS	Linux (Fedora Core 8)	Linux (Fedora Core 9)

4.1 Methodology

All the proposed designs have been implemented on a Xilinx FPGA device and synthesized by the Xilinx ISE 12.3 package. The XC5VLX110T, which is one of the mid-range devices in the modern Virtex-5 family, was chosen as the target device. This device provides 17,280 Virtex-5 slices and a maximum of 5,328 Kb of BlockRAM blocks [Xilinx 2009b]. The module was evaluated by performing the post-place and route simulation using ModelSim SE 6.5e.

XC5VLX110T has been chosen since it can provide enough resource and performance considering practical network coding specifications. A coefficient size between 16 and 512 can be efficiently used in the applications of network coding and from the initial analysis, those matrix sizes can be implemented with XC5VLX110T. In addition, this device can be easily integrated in middle sized commercial devices, such as tablet PCs, laptop computers, and digital TVs.

To compare the performance of our design with that of the software approach, we have adopted the parallelized algorithm introduced by [Park et al. 2010] as a reference, and we have executed this on a commodity desktop with two different systems: one with the Intel Core 2 Quad Q9400 processor and the other with the AMD Phenom-X4 9550 processor. More detailed specifications of the processors and test environments are listed in Table II.

4.2 Synthesis Results

The Xilinx FPGA devices are composed of several functional blocks; however, the configurable logic blocks (CLBs) and BlockRAM blocks are the only ones of interest, since our design mainly utilizes these two components. A CLB is a group of slices and a slice, in the case of the Virtex-5 family, contains four lookup tables (LUTs) and four flip-flops [Xilinx 2009b].

Table III. Utilization of the FPGA Components

Size	INVF Lookup GF ALU			INVF Computational GF ALU			FGJ Lookup GF ALU			FGJ Computational GF ALU		
	Slice Flip-Flop	Slice LUTs	Block RAM	Slice Flip-Flop	Slice LUTs	Block RAM	Slice Flip-Flop	Slice LUTs	Block RAM	Slice Flip-Flop	Slice LUTs	Block RAM
16x16	881 (1%)	4,705 (6%)	180 (3%)	1,468 (2%)	9,093 (13%)	36 (1%)	667 (1%)	4,696 (6%)	180 (3%)	1,839 (2%)	9,584 (13%)	36 (1%)
32x32	1,596 (2%)	9,101 (13%)	360 (6%)	2,844 (4%)	18,076 (26%)	72 (1%)	1,197 (1%)	8,344 (12%)	360 (6%)	3,791 (5%)	17,518 (25%)	72 (1%)
64x64	3,027 (4%)	18,346 (26%)	720 (13%)	5,346 (7%)	35,675 (50%)	144 (2%)	2,304 (3%)	11,784 (17%)	2,016 (37%)	6,956 (10%)	31,785 (46%)	144 (2%)
128x128	5,783 (8%)	39,977 (57%)	1,440 (27%)	10,481 (15%)	70,187 (101%)	288 (5%)	5,314 (7%)	28,529 (41%)	2,808 (52%)	11,766 (17%)	61,587 (89%)	288 (5%)
256x256	-	-	-	-	-	-	8,443 (12%)	26,169 (37%)	4,608 (86%)	14,027 (20%)	48,600 (70%)	4,608 (86%)
512x512	-	-	-	-	-	-	8,449 (12%)	26,292 (38%)	4,608 (86%)	13,600 (19%)	48,139 (69%)	4,608 (86%)

* Numbers in parentheses are percentages of source utilization

The utilization of the components for each matrix size and decoding method is listed in Table III and Fig. 9. Over the various designs with different sizes, the utilization of Slice Flip-Flops ranges from 1% to the maximum of 20%. Likewise, the utilizations of Slice LUTs and Block RAM range from 6% to 101% and from 1% to 86%, respectively. The design using the FGJ method exhibits the least area overhead. As explained in the previous section, the FGJ method eliminates the sub-modules for multiplications, which implies that we need less control logic as well. Conversely, the INVF method requires additional logic to control the buffer in the functional module and results in the highest resource utilization.

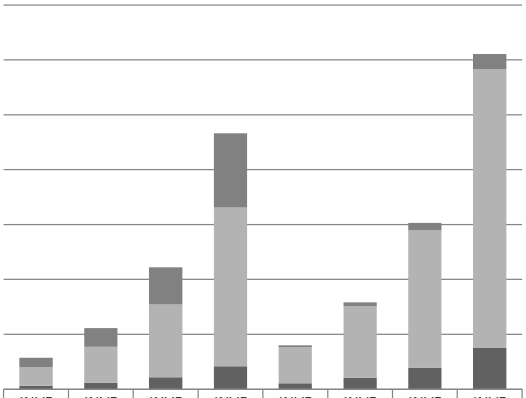
Each GF ALU still requires a sizeable area, which means that increasing the number of GF ALUs causes a significant area overhead. For instance, we could not synthesize 128 × 128 INVF with a computational GF ALU module since the area overhead was excessive and resulted in more utilization than 100%.

The area use of FPGA devices is also affected by the type of GF ALU. The decoders with the computational logic use about twice as many Slice LUTs. However, BlockRAMs use is considerably reduced compared to that of the lookup table method. To perform reciprocal logic and multiplication logic, the computational logic needs a considerable number of Slice LUTs but hardly uses memory space. In contrast, the GF ALUs using lookup table logic need a large number of memory components to store logarithm tables and the antilogarithm table.

According to the FPGA used and the network coding specification, selection between computation logic and lookup table for GF ALUs can be decided. If the area utilization is a more important factor, the lookup table can be used. On the contrary, when the decoding speed and the power efficiency are major concern, the computation logic can be utilized.

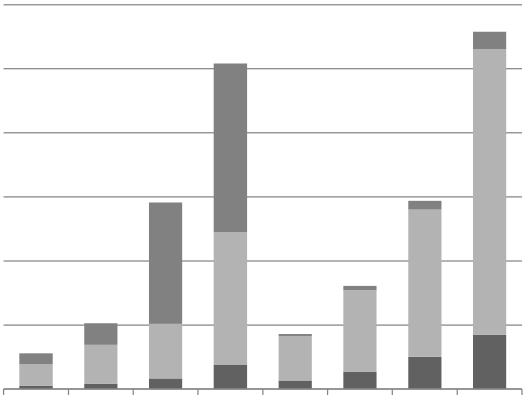
Each GF ALU can be chosen by either decoding method, INVF or FGJ, depending on the required decoder specification. However, each GF ALU still requires a certain area, which means that increasing the number of GF ALUs causes a significant area overhead.

When designing for larger coefficient vectors, the resource utilizations of the two sizes are very close. Since the structure is changed to use the microcode for the control, the size of the coefficient matrix does not affect the complexity of the control logic. Further, the control logic for each functional sub-module becomes simpler, so the two distinct decoding methods do not result in a significant difference of hardware overhead.



	INVF Lookup 16x16	INVF Lookup 32x32	INVF Lookup 64x64	INVF Lookup 128x128	INVF Comp. 16x16	INVF Comp. 32x32	INVF Comp. 64x64	INVF Comp. 128x128
BlockRAM	3.4%	6.8%	13.5%	27.0%	0.7%	1.4%	2.7%	5.4%
Slice LUTs	6.8%	13.2%	26.5%	57.8%	13.2%	26.2%	50.2%	101.5%
Slice Flip-Flop	1.3%	2.3%	4.4%	8.4%	2.1%	4.1%	7.7%	15.2%

(a)



	FGJ Lookup 16x16	FGJ Lookup 32x32	FGJ Lookup 64x64	FGJ Lookup 128x128	FGJ Comp. 16x16	FGJ Comp. 32x32	FGJ Comp. 64x64	FGJ Comp. 128x128
BlockRAM	3.4%	6.8%	37.9%	52.7%	0.7%	1.4%	2.7%	5.4%
Slice LUTs	6.8%	12.1%	17.0%	41.3%	13.9%	25.3%	46.0%	89.1%
Slice Flip-Flop	1.0%	1.7%	3.3%	7.7%	2.7%	5.5%	10.1%	17.0%

(b)

Fig. 9. Resource utilization of (a) INVF decoders and (b) FGJ decoders.

4.3 Simulation Results

In this part, the performance evaluation and power estimation results of the proposed designs is presented. For the performance evaluation purpose, an aggressively parallelized software algorithm has been implemented and tested on two different desktop systems: one with an Intel Core 2 Q9400 quad-core processor and the other with an AMD Phenom-X4 9550 quad-core processor. The main reason to compare with the high-end general purpose processors is that if the proposed accelerator results in better throughputs, then it is obvious that the throughputs are better than that achieved in embedded systems. To fully exploit the available parallelism of multi-core processors, we have used a parallelized decoding algorithm, known as DVP [Park et al. 2010]. Regarding the proposed FPGA approaches, we

have performed full simulations on Xilinx ISE 12.3 and XPower Analyzer [Xilinx 2010]. With the post-compilation results, the maximum possible clock frequency was obtained as 83.3 MHz.

Software algorithm for decoding process is implemented and tested on an ARM Cortex-A9 processor for power consumption comparison. We used a Tegra 2 processor, working at the operating frequency of 1 GHz, and the code for decoding process is cross-compiled by the GNU ARM cross compiler with the –O3 optimization level and executed on the Linux environment. To estimate actual power consumption of decoding process, both idle power consumption and decoding power consumption have been measured.

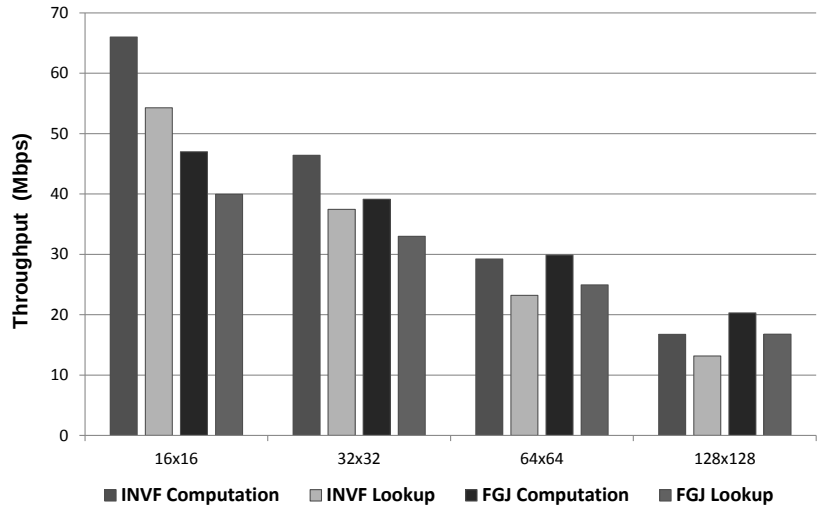
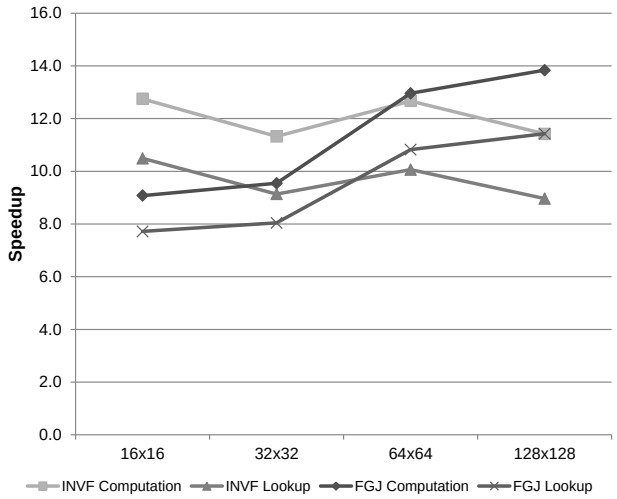


Fig. 10. Throughputs of the proposed FPGA designs for smaller sizes of coefficient matrix.

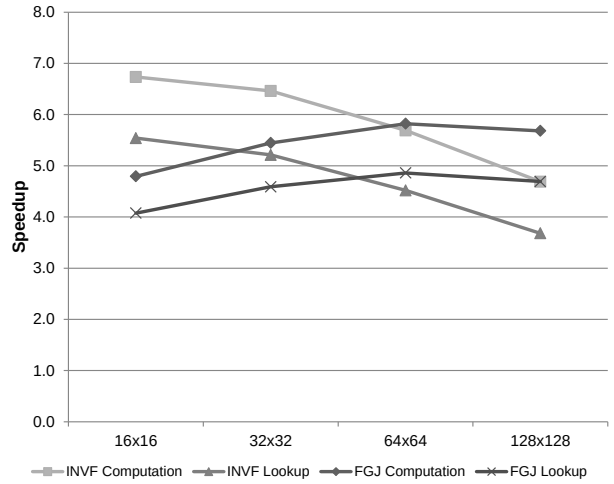
The performance results with the designs for the four different coefficient vectors are shown in Fig. 10. The results are given in terms of throughput. We have measured the execution delay of the entire decoding procedure for the encoded data matrix; therefore, the throughput has been calculated based on the data size and the encoding delay. In the case of the 16 × 16 size, the FPGA design for INVF with computational logic yields a throughput of 65.98 Mbps and that for FGJ with computational logic yields 46.98 Mbps, while the corresponding designs with lookup tables yield 54.28 Mbps and 39.94 Mbps. With the same coefficient matrix, the DVP algorithm on the AMD and Intel systems yields 5.18 Mbps and 9.80 Mbps, respectively. In the case of the 128 × 128 size, the maximum throughput of 20.30 Mbps is achieved with the FGJ design using computational logic. Note that 1.47 Mbps and 3.57 Mbps are achieved by the DVP algorithm on the two desktop systems. Considering that the proposed architecture is aimed at providing realistic performance for portable hand-held devices such as smart phones and tablet PCs, these results are acceptable for practical use.

Fig. 11 shows the speedup of the four FPGA designs over the DVP algorithm run on the AMD and Intel machines. The results show that the design using the FGJ method with computational logic gives the better performance as the coefficient matrix becomes larger. When the coefficient vector is as small as 16 elements, a speedup of 12.75 (compared to AMD) and a speed up of 6.73 (compared to Intel) have been achieved using the INVF model with computational logic. However, the INVF model exhibits decreased speedup when decoding larger sizes of coefficient matrix, whereas the FGJ model exhibits increased speedup. In the case of the 128 × 128 size,

the FGJ model with computational logic yielded the best speedup of 13.84 (compared to AMD) or 5.68 (compared to Intel).



(a) Results for speedup compared to AMD Phenom-X4 9550 processor



(b) Results for speedup compared to Intel Core 2 Quad Q9400 processor

Fig. 11. Speedup results of the proposed FPGA designs

Compared to the DVP algorithm run on both AMD and Intel machines, INVF and FGJ tend to yield higher performance. Although the speedup by the INVF method decreases rapidly compared with that by the FGJ method, the INVF model still results in very good performance, so the performance results of all four designs show promise. In contrast to the DVP algorithm, the proposed designs can maintain a high degree of parallelism by incorporating more GF ALUs and can provide better throughput.

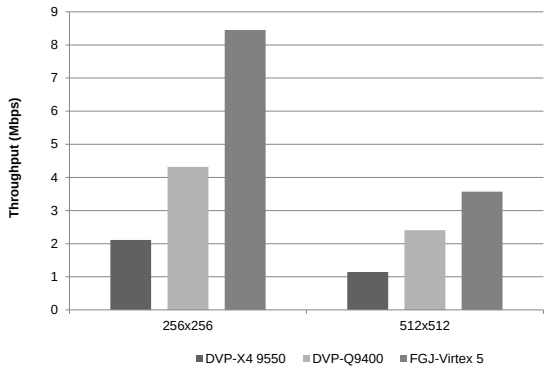


Fig. 12. Throughputs of the proposed FPGA design for larger sizes of coefficient matrix.

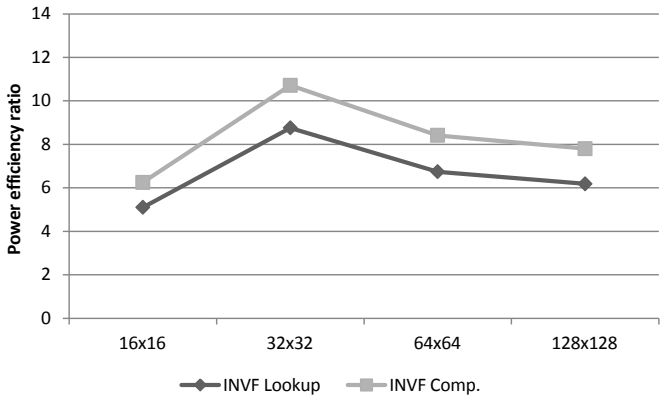
For coefficient vectors larger than 128, we have used 128 GF ALUs as described in Section 3.4 and chose the FGJ design since it shows better performance when decoding large coefficient matrices and thus only the results of the FGJ design is depicted in Fig. 12. Again, the baseline performances are given as the results of the DVP algorithm. With sizes of 256 and 512 the FGJ design yields throughputs of 8.45 Mbps and 3.57 Mbps, respectively, while the DVP algorithm yields 2.11 Mbps and 1.14 Mbps on the AMD system or 4.32 Mbps and 2.41 Mbps on the Intel system.

Unfortunately, the tendency for an increasing performance gap between the FPGA designs and the DVP algorithm has not been observed. This is due to the fact that we did not further increase the number of GF ALUs in this design. However, our design still yields better performance than the parallelized software algorithm, and this is very promising as we target reconfigurable network coding accelerators that can be used in various embedded systems.

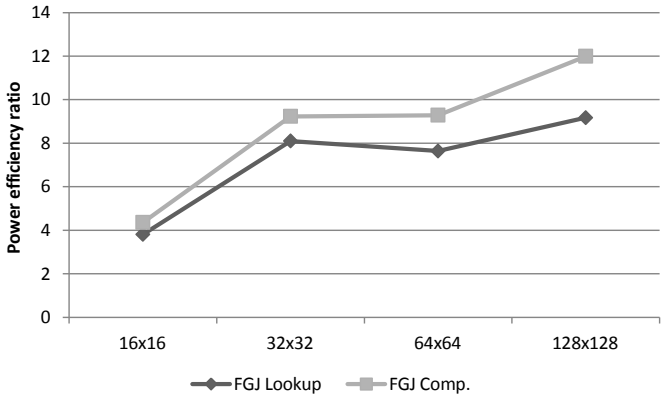
Table IV. Power Efficiency of ARM Processor and Comparison

Size	ARM power efficiency (Mbps/Watt)	FPGA Design power efficiency (Mbps/Watt)			
		INVF Lookup	INVF Comp.	FGJ Lookup	FGJ Comp.
16x16	8.40	42.87	52.53	32.00	36.59
32x32	3.20	28.03	34.30	25.92	29.55
64x64	2.22	14.98	18.69	16.97	20.64
128x128	1.11	6.87	8.66	10.18	13.31

The power estimation results with the design are presented in Table IV and Fig. 13. Power efficiency of FPGA design is estimated when the design makes its maximum throughput. It can be varied by operating frequency. We have compared the throughput per watt of our designs and ARM Cortex-A9 processors in terms of power efficiency, the ratio of Mbps per Watt FPGA to ARM processor. The results show that FPGA design is power efficient than ARM processor. In small coefficient vectors, power efficiencies of INVF models are higher than FGJ models. With 32 × 32, the INVF computational model shows a maximum of 10.7 times higher power efficiency ratio whereas the FGJ computational model shows 9.2. When coefficient vectors become larger, however, the FGJ computational model with 128 × 128 shows 12 times higher power efficiency ratio compared to the ARM processor, which is far better than INVF computational model. From the results, it has been shown that the proposed network coding accelerator provides high decoding performance and that consequently increases the power efficiency.



(a) Power efficiency ratio of INVF model



(b) Power efficiency ratio of FGJ model

Fig. 13 Power efficiency ratio of the FPGA design to ARM processor

5. CONCLUSIONS

A high-performance decoder for network coding has been proposed using FPGA technology that can be widely adopted in various embedded systems. A parallelized structure of GF ALUs has been developed to maximize the throughput. Two different types of decoder have been proposed and each decoder has been designed to use two different calculation methods in order to reveal trade-offs in terms of area and performance. For the larger sizes of the coefficient matrix, two additional designs have been introduced. Those have been compared to a parallelized network coding approach executed on both the AMD and Intel quad-core processor systems. Also, power efficiency of the design compared to an ARM Cortex-A9 processor.

Considering four different coefficient vector sizes of 16, 32, 64, and 128 elements, the proposed design provides a maximum speedup of 13.84 compared to a quad-core AMD processor and 6.73 compared to a quad-core Intel processor. Note that all the software implementations on the general-purpose processors were aggressively parallelized. Moreover, in power efficiency, the proposed design affords a maximum 12 times higher throughput per watt than an ARM processor.

The proposed architecture mainly targets small embedded systems. Therefore, this performance improvement over the high-end commercial microprocessor encourages development of our FPGA-based network coding method. As mentioned above, this would provide both programmability and performance. Furthermore,

high power efficiency of our design could make FPGA accelerators more suitable for embedded systems.

As an extension of this research, we will further investigate the possibility of increasing the maximum working frequency and minimizing the usage of FPGA components by optimizing the control logic and reducing the area of each functional unit to achieve higher performance and high adaptability to small mobile devices.

REFERENCES

AHLSWEDE, R., CAI, N., LI, S. R., AND YEUNG, R. W. 2000. Network Information Flow. *IEEE Trans. Information Theory*, vol. 46, no. 4. 1204-1216

CHACHULSKI, S., JENNINGS, M., KATTI, S., AND KATABI, D. 2007. Trading Structure for Randomness in Wireless Opportunistic Routing. In *Proceedings of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Kyoto, Japan, Aug. 2007.

CHELTON, W. N. AND BENAISSA, M. 2008. Fast Elliptic Curve Cryptography on FPGA. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, 198-205.

CHOU, P. A., WU, Y., AND JAIN, K. 2003. Practical Network Coding. In *Proceedings of 41st Allerton Conference of Communication, Control and Computing (Allerton '03)*, Monticello, IL, USA, Oct. 2003.

CHOU, P. A. AND WU, Y. 2007. Network Coding for the Internet and Wireless Networks. *IEEE Signal Processing Magazine*, vol. 24, no. 5, 77-85.

DEBIAN, http://www.debian.org

DESCHAMPS, J.-P. AND SUTTER, G. 2006. Hardware Implementation of Finite-Field Division. *Acta Applicandae Mathematicae*, vol. 93, no. 1-3, 119-147.

GULATI, K. AND KHATRI, S. P. 2008. Improving FPGA Routability Using Network Coding. In *Proceedings of 18th ACM Great Lakes Symposium on VLSI*, Orlando, FL, USA, May 2008.

HO, T., MÉDARD, M., KOETTER, R., KARGER, D. R., EFFROS, M., SHI, J., AND LEONG, B. 2006. A Random Linear Network Coding Approach to Multicast. *IEEE Trans. Information Theory*, vol. 52, no. 10, 4413–4430.

ITOH, T. AND TSUJII, S. 1988. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) using Normal Bases. *Information and Computation*, vol. 78, no. 3, 171–177.

KATTI, S., KATABI, D., BALAKRISHNAN, H., AND MÉDARD, M. 2008. Symbol-level Network Coding for Wireless Mesh Networks. *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. 401-412

KIM, C. H. AND HONG, C. P. 2002. High-speed division architecture for GF(2^m). *Electronics Letters*, vol. 38, no. 15, 835–836.

KIM, D., PARK, K., AND RO, W. W. 2011. Network Coding on Heterogeneous Multi-Core Processors for Wireless Sensor Networks. *Sensors*, vol. 11, no. 8, 7908-7933

KIM, D., PARK, K., AND RO, W. W. 2012. Exploiting SIMD Parallelism on Dynamically Partitioned Parallel Network Coding for P2P Systems. *Computers and Electrical Engineering,* Available online 12 March 2012, ISSN 0045-7906, 10.1016/j.compeleceng.2012.02.009.

KIM, S. AND RO, W. W. 2012. Reconfigurable and Parallelized Network Coding Decoder for VANETs. *Mobile Information Systems,* vol.8, no. 1, 45-59.

KIM, S. AND RO, W. W. 2010. FPGA Implementation of Highly Parallelized Decoder Logic for Network Coding. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* Monterey, CA, USA, Feb. 2010, 284-284.

KOETTER, R. AND MÉDARD, M. 2003. An algebraic approach to network coding. *IEEE/ACM Trans. Networking*, vol. 11, no. 5, 782-795.

LEE, S. AND RO. W. W. 2010. Accelerated Network Coding with Dynamic Stream Decomposition on Graphics Processing Unit. *The Computer Journal*, vol. 55, no. 1, 21-34

LI, S. -Y. R., YEUNG, R. W., AND CAI, N. 2003. Linear network coding. *IEEE Trans. Information Theory*, vol. 49, no. 2, 371-381.

LIN, S. AND COSTELLO JR., D. 1983. *Error Control Coding: Fundamentals and Applications*. Prentice Hall Englewood Cliffs, NJ.

MACE, F., STANDAERT, F. X., AND QUISQUATER, J. J. 2008. FPGA Implementation(s) of a Scalable Encryption Algorithm. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, 212-216.

MASTROVITO, E. 1991. VLSI Architectures for Computation in Galois Fields. *Ph.D thesis*, Dept. of Electrical Eng., Linkoping Univ., Sweden.

MEHROTRA, P., SINGHAI, M., PRATT, M., CASSADA, M., AND HAMILTON, P. 2004. FPGA Implementation of a High Speed Network Interface Card for Optical Burst Switched Networks. In *Proceedings of ACM/SIGDA 12th International Symposium on FPGA*, Monterey, CA, USA, Feb. 2004, 255-255.

NAGARAJAN, A., SCHULTE, M.J., AND RAMANATHAN, P. 2010. Galois field hardware architectures for network coding. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, San Diego, CA, USA, Oct. 2010

PARK, K., PARK, J. S., AND RO, W. W. 2009. Efficient Parallelized Network Coding for P2P File Sharing Applications. In *Proceedings of 4th International Conference on Grid and Pervasive Computing (GPC '09)*, Geneva, Switzerland, May 2009.

PARK, K., PARK, J. S., AND RO, W. W. 2010. On Improving Parallelized Network Coding with Dynamic Partitioning. *IEEE Trans. Parallel and Distributed Systems*. vol. 21, no. 11, 1547-1560.

PEDERSEN, M. V. F., FITZEK, H. P., AND LARSEN, T. 2008. Implementation and Performance Evaluation of Network Coding for Cooperative Mobile Devices. In *Proceedings of IEEE International Conference on Communications Workshops (ICC '08)*, Beijing, China, May 2008.

SHOJANIA, H. AND LI, B. 2007. Parallelized Progressive Network Coding with Hardware Acceleration. In *Proceedings of IEEE International Workshop on Quality of Service*, Evanston, IL, USA, Jun. 2007, 47-55.

SHOJANIA, H., LI, B., AND WANG, X. 2009a. Nuclei: GPU-accelerated Many-core Network Coding. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM2009)*, Rio de Janeiro, Brazil, Apr 2009, 459-467.

SHOJANIA, H. AND LI, B. 2009b. Pushing the Envelope: Extreme Network Coding on the GPU. In *Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS 2009)*, Montreal Canada, June 2009, 22-26.

SHOJANIA, H. AND LI, B. 2009C. Random Network Coding on the iPhone: Fact or Fiction? In *Proceedings of 18th international Workshop on Network and Operating Systems Support for Digital Audio and Video*, Williamsburg, VA, USA, Jun 2009, 37-42.

XILINX, San Jose, CA, 2009a. Block Memory Generator v.4.3 Product Specification. http://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen_ds512.pdf.

XILINX, San Jose, CA, 2009b. Virtex-5 Family overview. http://direct.xilinx.com/bvdocs/publications/ds100.pdf.

XILINX, San Jose, CA, 2010. ISE Design Suite Software Manuals and Help. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/manuals.pdf.

YAN, Z., AND SARWATE, D. V. 2003. New systolic architectures for inversion and division in GF(2^m). *IEEE Trans. Computer*, vol. 52, no. 11, 1514–1519.

YEUNG, R. W. AND ZHANG, Z. 1999. Distributed Source Coding for Satellite Communications. *IEEE Trans. Information Theory*, vol. 45, no. 4, 1111-1120.

YOON, T. AND PARK, J. 2010. FPGA Implementation of Network Coding Decoder. *International Journal of Computer Science and Network Security,* vol.10 no.12, 34-39