SAND 2013-1418 C

Speed Optimized Implementations of the QUAD Algorithm

Jason Hamlet and Robert Brocato

Sandia National Laboratories

jrhamle@sandia.gov, rwbroca@sandia.gov

Abstract

We present several software and hardware implementations of QUAD, a recently introduced stream cipher designed to be provably secure and practical to implement. The software implementations target both a personal computer and an ARM microprocessor. The hardware implementations target field programmable gate arrays. The purpose of our work was to first find the baseline performance of QUAD implementations, then to optimize our implementations for throughput. Our software implementations perform comparably to prior work. Our hardware implementations are the first known implementations to use random coefficients, in agreement with QUAD's security argument, and achieve much higher throughputs than prior implementations.

Keywords: *QUAD, stream cipher, throughput optimization, hardware acceleration*

1. Introduction

The QUAD algorithm is a stream cipher proposed by Berbain, Gilbert, and Patarin and is intended to be provably secure and practical to implement [1]. QUAD's security is derived from the difficulty of solving the multi-variate quadratic (MQ) problem. That is, the security of the QUAD cipher is provably reducible to the NP-hard problem of finding a solution to a multivariate quadratic system of *m* quadratic equations in *n* variables over a finite field, *GF(q)*. In this paper, we report on our efforts to measure computational performance of the QUAD algorithm on a personal computer (PC), an ARM Cortex A8 embedded microprocessor, and an Altera Cyclone V field programmable gate array (FPGA). We implemented the QUAD algorithm with a number of different variations on each platform in an effort to optimize performance on each. The standard measure of performance that we seek to optimize throughout these tests is the rate of keystream production, measured in bits/second.

Each equation in the system of *kn* multivariate quadratic equations used in QUAD is written as

$$Q(x) = \sum_{1\le i\le j\le n} \alpha_{i,j} x_i x_j + \sum_{1\le i\le n} \beta_i x_i + \gamma \tag{1}$$

In this work we consider only *n=128, k=2,* and coefficients in *GF(2)*. While implementations of QUAD for fields larger than GF(2) are possible, they have been shown to be insecure [2]. The solution to equation (1) is the *m=256* bit value $S(x)=(Q_1(x), \ldots, Q_{256}(x))$. Values $S_{out}(x)=(Q_{129}(x),\ldots,Q_{256}(x))$ are output as the keystream, and values $S_{it}(x)=(Q_1(x),\ldots,Q_{128}(x))$ are used to update the internal state. The α, β, and γ coefficients are random but public values. There are $256\left(\binom{128}{2}\right) = 2080768$ bits of α, which we term nonlinear coefficients, $256 \times 128 = 32{,}768$ bits of linear coefficients β, and 256 bits of γ, for a total of 2,113,792 bits. QUAD's security argument requires these coefficients to be chosen randomly, and its

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

designers state that “bad” choices of coefficients are unlikely, though this has not been proven [1]. The coefficients used in our implementations were generated using the random number generator in OpenSSL [3].

2. Software Implementations

Using the C programming language, we implemented four different software versions of QUAD. Each version used the same random coefficients. For each version we targeted both a personal computer and an ARM microprocessor and measured the resulting throughputs, which vary significantly between implementations. In this section we describe each of the implementations and our results.

2.1. Software Implementation: Overview

The initialization used in all of our software and hardware implementations of the QUAD algorithm differs from that presented in [1]. The original Eurocrypt paper describes an initialization procedure that uses two different multivariate quadratic systems, S_0 and S_1, of *n* equations in *n* unknowns. The internal state *x*, which has been set to an initial value *K*, is used to select either the output of $S_0(x)$ or of $S_1(x)$, depending on the sequentially selected value of the internal state. However, this approach is non-standard and has been removed from cryptographic standards that include QUAD [4]. For our software implementations we simply make a call to the Unix function */dev/random* to provide an entropy source to seed the internal state of the algorithm. In practice, one might concatenate the output of an entropy source with a personalization string and then apply a cryptographic hash function to the result to seed the internal state of the algorithm. For the purpose of this work initialization approaches are inconsequential, since we have removed the effects of that delay from our throughput time measurements. Consequently, the initialization can be viewed as an initial delay that is identical between implementations and whose impact decreases as the length of the generated keystream increases.

Each of our software implementations uses a 256-bit internal state with a 128-bit keystream output for each update cycle. We tested these programs on both a PC and an ARM microprocessor. The PC used for testing has a 3.0GHz Intel Core 2 Duo processor with 6 Mbytes of cache memory and 8 Gbytes of random access memory (RAM) running Red Hat Enterprise Linux version 5. We compiled our code with the Gnu compiler version 4.1.2. We also ran each program on a Cortex A8 ARM core that is part of a DaVinci DM3730 microprocessor. The ARM microprocessor is almost certainly the most commonly used microprocessor in the world, operating in 95% of all smartphones, 90% of all hard disk drives, and 40% of all digital televisions and set-top boxes [5]. The DM3730 microprocessor has an additional C64x digital signal processor (DSP) core. We attempted to compile our programs to run on the DSP core, but we had insufficient development time to replace the key C language functions used to measure algorithm execution times. Consequently, our reported speeds are limited to the PC and the ARM microprocessor.

2.2. Software Implementation: Results

2.2.1. QUAD1

Our first software implementation, QUAD1, computes the internal state value and keystream output by means of the most computationally simplistic approach. It was used to derive test vectors for the other software implementations. In this version, separate computations are performed to update the 128-bit internal state register and the 128-bit keystream output register. That is, the 256-bit internal state is treated as two separate registers in QUAD1. Computations for the nonlinear, linear, and constant terms are performed separately. No effort was made to streamline computations in QUAD1, and the bit-wise computations required for the QUAD algorithm are not well suited to the register-based computations of a PC running a C program. Due to these factors, this first software version of QUAD achieves an average speed of only 4.7 kbits/sec on the PC and 970 bits/sec on the ARM microprocessor.

2.2.2. QUAD2

Our second version, QUAD2, was created in an effort to speed up the implementation of the algorithm by performing block matrix-vector multiplication. Most of the computations required in the algorithm take place in the quadratic, $\alpha_{ij}x_ix_j$, terms. To speed up these computations, the arithmetic in QUAD2 is performed on words sized to fit the register size in the microprocessor, which is 32 bits for the ARM and 64 bits for the Intel Duo. This version was also designed to be easily implemented on a smaller microprocessor with 8-bit or 16-bit registers. A set of appropriately sized Hadamard transform functions were created that implement a parity popcount to count the number of ones in the input register. The arithmetic is then performed using these and a bitwise logical AND function. QUAD2 achieved an average speed of 34.6 kbits/sec on the PC and 5.1 kbits/sec on the ARM processor.

2.2.3. QUAD3

A third software version, QUAD3, was created as a specialization of QUAD2. This version targets the 32-bit architecture found on the ARM processor. In this version, all of the computations are routed through a parity function that can be implemented in a DSP hardware instruction for a 32-bit parity popcount. This version ran at an average speed of 111 kbits/sec on the PC and 16.2 kbits/sec on the ARM processor.

2.2.4. QUAD4

Our fourth software version of QUAD, named QUAD4, was created by moving as much computation as possible outside of the loops that are used to compute each coefficient. It makes use of a custom logical XOR inline function to XOR all bits in a register. The basic version of QUAD4 generated an average of 1.25 Mbits/sec of keystream output on the PC and 163 kbits/sec on the ARM. Versions of the QUAD4 implementation, named QUAD4_pc_32bit and QUAD4_pc_64bit, were created with relatively minor changes to C function implementations. These provided little speed improvement over the basic QUAD4 program.

Versions of QUAD4 named QUAD4_dsp, QUAD4_dsp2, and QUAD4_dsp3 were created to progressively remove more high level C-language functions and replace them with low-level implementations. This was done to enable the QUAD algorithm to run on the C64x digital signal processor core that is present in the DM3730 microprocessor, along with the ARM microprocessor. It was possible to get the program to execute on the C64x core; however, the C-language function *time,* used to benchmark program performance, does not function in the C64x. No suitable replacement for this function was able to be created in the available development time. As a result, the DSP optimized versions of QUAD were only tested on the PC and on the ARM processor.

In QUAD4_dsp the coefficient table was moved directly into program memory from the external file that was used in previous versions. Also, the entropy source used for initialization was hard-coded into program memory. These two actions approximately tripled the throughput to 3.30 Mbits/sec on the PC and 547 kbits/sec on the ARM. QUAD4_dsp2 incorporates minor changes to output data handling. It achieves only minor performance improvements over QUAD4_dsp. In QUAD4_dsp3, the bit test used to decide the condition of the state bit was performed using a look-up table. This decreased throughput to 2.19 Mbits/sec in the PC and 451 kbits/sec in the ARM processor.

Throughput results for our software implementations are summarized in Table I. Our fastest performing version uses a custom, inline, register-based XOR function with coefficients stored in program memory and some high level C language functions replaced by low level operations. Implementation differences between the best version for the PC and the best version for the ARM microprocessor are minimal. Our fastest speeds on the PC were slightly slower than those reported in [1]. This is to be expected since that implementation uses a 160-bit internal state with 80-bit output versus our 256-bit internal state with 128-bit output. Adjusting for the size difference in internal state and output, the speed of our fastest implementation of QUAD is roughly equal to that reported in [1].

Table I. Throughput results for our software implementations of QUAD

	Linux PC (Mb/s)	**ARM μP (Mb/s)**
QUAD1	0.0047	0.00097
QUAD2	0.0364	0.0051
QUAD3	0.111	0.0162
QUAD4	1.21	0.163
QUAD4_pc_32bit	0.845	0.132
QUAD4_pc_64bit	1.25	N/A
QUAD4_dsp	3.302	0.547
QUAD4_dsp2	3.298	0.582
QUAD4_dsp3	2.190	0.451

3. Hardware Implementations

3.1. Hardware Implementation: Overview

There has been some previous work on area efficient hardware implementations of QUAD [4, 5], but that work abandons the formal security proof of [1] by generating the function *S* pseudo-randomly. Additionally, while area efficient, the designs in [4, 5] provide maximum throughput of 4.1Mbps, which is not significantly faster than our software implementations. In this work, we present hardware FPGA implementations of QUAD that retain random functions *S* and that achieve much higher throughput in a reasonable physical area.

We describe two FPGA implementations of QUAD. To achieve high-throughput, area-efficient implementations, both of these hardware designs are tailored specifically for the Cyclone V FPGA architecture [6]. The target device primarily impacts memory layout and the specifics of look-up table (LUT) based combinatorial logic. Similar results could be achieved with other FPGA architectures, and suggestions for those architectures are provided. Though it results in less portable hardware development language (HDL) code, attention to FPGA architecture allows faster, smaller designs.

As with the software implementations, the hardware implementations of QUAD that we describe do not include the key initialization procedure described in [1]. Instead, the state is initialized to a constant. Our hardware designs are over GF(2) with key length *n=128* and *k=2*, resulting in a set of *m=kn=256* equations. Other key lengths in *GF(2)* would have similar designs. Implementations over larger finite fields are possible, but are much less secure and so are of less practical interest [2]. We do not consider them here.

The *Q(x)* in equation (1) have a regular structure that allows many different hardware implementations. For instance, the *Q(x)* could be solved serially. If a serial design requires ζ cycles to compute one of the *Q(x)* then such an approach would need $\zeta \times n \times k$ cycles to generate the *nk*-bit result. If η of the *Q(x)* are computed in parallel then $\frac{\zeta}{\eta} \times n \times k$ cycles would be needed, but the hardware cost would also increase roughly by a factor of η.

In our designs we divide the *Q(x)* terms into a nonlinear portion $\sum_{1\le i\le j\le n} \alpha_{i,j} x_i x_j$ and a linear portion $\sum_{1\le i\le n} \beta_i x_i$ that are computed separately and then combined with γ. Our first design splits computation of the *Q(x)* equations into two stages. In each stage, 128 of the *Q(x)* terms are computed in parallel. The same combinatorial logic and memory resources are used in the two stages. As such, the processing units described in hardware design sections each appear 128 times in the first design. The second design computes all 256 of the *Q(x)* in parallel and requires 256 instantiations of the processing units. Since logic reuse is not possible in this design, it is larger, but it has a greater throughput. The basic structure of these designs is shown in Figure 1, which depicts the data path for computing one of the *Q(x)* terms. Duplication of the linear and nonlinear processing units is required to compute the *Q(x)* terms in parallel.

2.1. Nonlinear Computation: Combinations of State

Computation of the nonlinear portion requires generation of the $\binom{n}{2} = \binom{128}{2}$ combinations of the current state, x, multiplying each of these combinations by the appropriate $\alpha_{i,j}$, and finally performing a summation of these $\binom{128}{2}$ values to a single bit. First, we consider generating the combinations of the current state, that is, generating x_ix_j, $1 \le i \le j \le n$. For state size *n,* a straightforward approach is to generate *n* bits per cycle. This can be accomplished with two registers. Initially, one register holds the current state, *x*, and the second register holds x rotated by one bit. The contents of the two registers are pair-wise ANDed to produce the combinations x_ix_j. Then the second register is rotated by a single bit and the operation is repeated. For *n=128*, it will take 64 cycles to generate the $\binom{128}{2}$ combinations. On the last cycle, the upper 64 bits of the result will hold the same combinations as the lower 64 bits. This can be corrected by masking the duplicate combinations with zeros in the $\alpha_{i,j}$ coefficients. By making the registers smaller than *n* bits wide, fewer combinations can be generated each cycle, although this requires the two registers to be periodically updated with new portions of the current state. More than *n* bits can be generated in each cycle by using more registers. For instance, we could use three registers, two of them initially containing *x* and the third containing *x* rotated by *n/2* bits. We could then generate *2n* combinations in each cycle. In this approach the second and third registers would each be rotated by one bit each cycle. Straightforward extensions to this scheme would allow more combinations to be generated each cycle. Our FPGA designs both have 128-bit data paths. They use two 128-bit registers, and so require 64 cycles to generate all of the combinations.

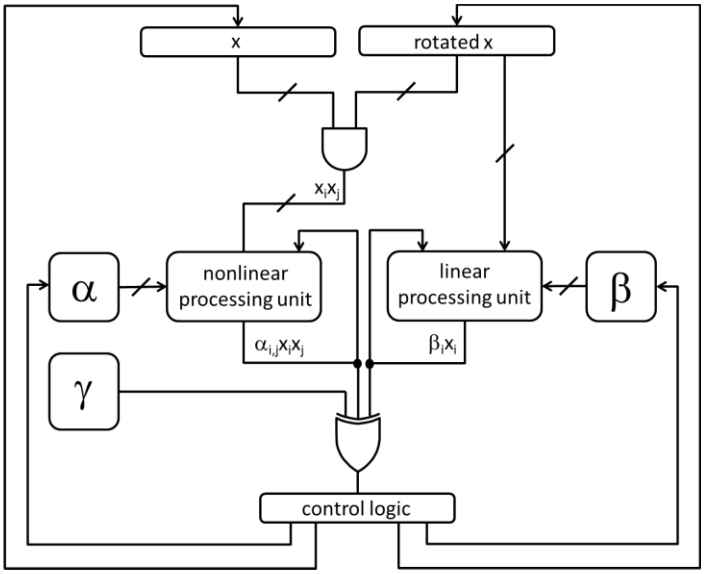

Figure 1. At the top level, our design has separate linear and nonlinear processing units that operate on the function as the current state. The design is readily parallelized by replicating these processing units.

2.2. Nonlinear Coefficients

Each of the *Q(x)* equations requires $\binom{128}{2} = 8128$ $\alpha_{i,j}$ values. It is convenient to store these either in a ROM with output word width equal to the number of combinations of x_ix_j, generated each cycle, or in a collection of ROMs whose combined output word width equals this number of combinations. A more detailed discussion of ROM geometries appears in Section 2.5. Just as generation of the x_ix_j combinations will generally produce some redundant combinations, the chosen ROM geometries are likely to store more than 8128 bits. The unused bits should be set to zeros so that they will mask the redundant x_ix_j. Additionally, the $\alpha_{i,j}$ should be permuted prior to storage to compensate for the ordering of the generated sequence of x_ix_j combinations.

2.3. Combining the Nonlinear Coefficients and Combinations of State

The $\binom{128}{2} = 8128$ combinations of state and nonlinear coefficients have to be bitwise multiplied and then summed to produce a single bit. In the straightforward approach, during each clock cycle the combinations of state generated by the registers are ANDed with the nonlinear coefficients. The results are then input to an XOR tree. This XOR tree will generally be several layers deep, so it can be pipelined to improve throughput. While simple, this approach does not consider the FPGA architecture or resources. More efficient designs are possible by tailoring the bitwise AND multiplication and wide XOR summation to the available FPGA resources.

Modern FPGAs consist of an array of reconfigurable units, each containing look-up tables (LUTs), memory elements, routing, and other resources such as multiplexors or adders. The particulars of these units vary by manufacturer and device family. The LUTs are used to implement user-defined logic functions. Area efficient designs should attempt to make full use of the LUTs. For example, if a device provides 4-1 LUTs then 1, 2, 3, and 4-input, 1-output functions all require one LUT. Partitioning the design into functions of fewer than 4 inputs makes inefficient use of the LUTs and wastes resources.

The Cyclone V devices targeted in this work contain 8 input fracturable LUTs [6]. Each of these LUTs can be configured to implement one 6-input function, any two 4-input functions, any combination of one 3-input function and one 5-output function, or various more complicated configurations with functions having shared inputs. To make efficient use of the FPGA resources, we organize our design to make full use of these LUTs. An overview of our nonlinear processing unit is shown in Figure 2a, which we will refer to for the remainder of this discussion. The AND-XOR LUT blocks each consist of a 6-1 LUT and each implement three of the multiplications and two of the summations required for the nonlinear processing. That is, each AND-XOR block in the nonlinear portion computes $f(x_i, \alpha_i) = \sum_{0 \le i \le 2} x_i \alpha_i$ where each of the x_i is the previously computed pairwise AND of two bits of state and the α_i are the corresponding nonlinear coefficients. Since our designs have a 128-bit data path there are 42 of these 6-input AND-XOR functions and one 4-input AND-XOR in each non-linear processing unit. The result is a 43-bit value that must be XORed down to 1 bit. For this, we have a pipelined architecture that again makes full use of the available LUT resources. The first stage of the XOR tree consists of seven 6-input LUTs, each of which computes a 6-bit wide XOR. Finally, the seven output bits from the first stage of the XOR tree, the 43rd input to the XOR tree, and the previous output from the XOR tree are combined by a

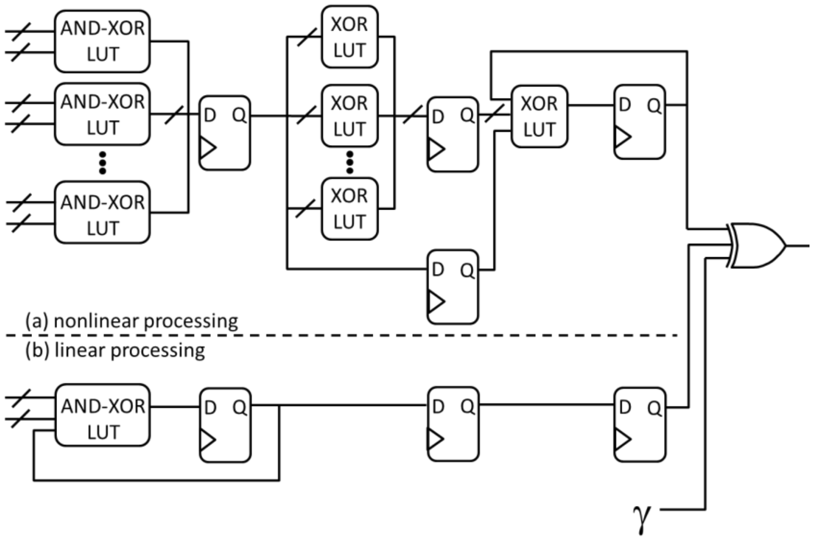

Figure 2. Detailed depiction of our pipelined, LUT-based nonlinear (top) and linear (bottom) processing units.

9-input XOR. The feedback path is required because each of the *Q(x)* terms requires operations on $\binom{128}{2}$combinations of state and nonlinear coefficients, and we have a 128-bit data path. Consequently, it takes 64 rounds of this processing flow for the nonlinear processing unit to generate a single output bit.

Now, we briefly consider implementation of the nonlinear processing in Virtex-4 devices, which were used in previous FPGA implementations [4]. Each Virtex-4 slice contains two 4-input LUTs [7]. To accommodate this structure, the AND-XOR LUTs in Figure 2a would be redesigned to implement $f(x_i, \alpha_i) = \sum_{i=0,1} x_i\alpha_i$ where each of the x_i is a previously computed pairwise AND of two bits of state and the α_i are the corresponding nonlinear coefficients. For our 128-bit data path, 64 AND-XOR LUTs would be required. The first stage of the XOR tree would consist of 16 4-input XOR LUTs. The second stage of the XOR tree would then combine these 16 bits, or a third stage could be added.

2.4. Linear Computation

The linear portion of QUAD is more easily computed than the nonlinear portion. Since we require 64 cycles to calculate the non-linear portion and there are 128 terms in the summation of the linear portion, we operate on two bits of state and two linear coefficients each clock cycle. In Figure 2a the AND-XOR LUT computes $g(x_i, \beta_i) = \sum_{i=0,1} x_i\beta_i$ where the x_i are taps off the shifted version of the current state. Our design uses the leftmost (127th) and 63rd bits of this register, but any two bits separated by 64 positions would be acceptable. As with the linear coefficients, the β_i coefficients should be permuted prior to storage to compensate for the utilized sequence of β_i. In our designs the ROMS that store the β coefficients are designed to make efficient use of the available memory resources. The details are provided in Section 2.5.

Virtex-4 devices could implement the AND-XOR LUT in Figure 2b in a single 4-input LUT. This would make more efficient use of the device resources than the Cyclone V design, which uses only four of the six available LUT inputs.

2.5. Memory layout

For QUAD implementations with state size *n=128* and *k=2* there are $256\left(\binom{128}{2}+128+1\right)$coefficients that must be stored. Given this large number of coefficients, it is important to choose ROM geometries that efficiently store the coefficients while allowing the full set of coefficients to be accessed in the number of cycles required by the pipelined architecture and throughput constraints. For designs targeting large throughputs many memories are required so that many coefficients can be accessed concurrently. Dual and quad-port memories are also useful in this regard.

Cyclone V devices contain two types of memory blocks [8]. The relevant features of these blocks are summarized in Table II. Each of the *nk=256* equations *Q(x)* requires storing $\binom{128}{2}=8128$ nonlinear coefficients, $\alpha_{i,j}$ *1≤i≤j≤n*, so a single M10K block is large enough to store the $\alpha_{i,j}$ for one of the Q(x). However, with a maximum configuration width of 40 bits, a dual port design would require at least 102 cycles to access all of the $\alpha_{i,j}$. To increase throughput several M10K blocks can be used. Unfortunately, this increase in throughput causes inefficient memory usage and increased resource consumption. Due to their small size, the MLAB memory resources in Cyclone V devices are not an attractive option for storing these coefficients.

Each of the *Q(x)* equations also requires storing 128 linear coefficients*,* β_i *1≤i≤n*. Four M10K blocks could be used to store all of the β_i. Using a dual port configuration, such an approach would require at least 103 cycles to access all of the β_i coefficients. Greater throughputs could be achieved by using more M10K blocks, but this would also result in inefficient use of the memories. MLAB resources are another option for storing the β_i. In the *32 x 16* configuration each MLAB can store the β_i coefficient for four of the *Q(x)* equations. This makes efficient use of the memory resources, and requires 32 cycles to read all of the β_i.

Since the $\alpha_{i,j}$ are used in calculating the nonlinear portion of the *Q(x)* terms, the β_i coefficients are used in finding the linear portion, and the results of these distinct portions are combined with the γ to produce *Q(x),* it is desirable for the linear and nonlinear portions to be calculated in parallel and for these calculations to take the same amount of time. To achieve this goal, the $\alpha_{i,j}$ and β_i coefficients should be stored so that it takes the same number of cycles to access them, and they should be stored in a manner that makes efficient use of the memory resources.

Table II. Memory resources available in Cyclone V devices

	M10K	**MLAB**
Configuration (depth x width)	256 x 32, 256 x 40, 512 x 16, 512 x 20, 1k x 8, 1k x 10, 2k x 4, 2k x 5, 4k x 2, 8k x 1	32 x 16, 32 x 18, 32 x 20
f_{max} (MHz)	315	450
Memory modes	Single port, true dual port, ROM	Single port, simple dual port, ROM

Our first design splits the calculation of the *Q(x)* equations into two rounds. The first 128 of the *Q(x)* are solved in the first round, and the remaining 128 are solved in the second. This permits the same memory resources to be used in each round. This design uses 128 memories to store the $\alpha_{i,j}$ coefficients. Each of these memories is composed of four M10K blocks, each 32 bits wide and 256 words deep. This was necessary to achieve the 128 bit wide memories necessary to support our pipeline, which requires 128 bits of new $\alpha_{i,j}$ values in each clock cycle. Unfortunately, this leaves half of each M10K block empty. Switching to a true dual port configuration does not improve the memory usage, as a 128 bit wide, 256 word deep dual-port ROM requires 8 M10K blocks. The memory layout for the 128 α coefficient memories is shown in Table III. This design uses 16 memories to store the β_i coefficients. Each memory is 16 bits wide by 128 words deep, completely fills four MLAB resources, and stores the β_i coefficients for 16 of the *Q(x)* equations. The memory layout is also shown in Table III. We chose this configuration so that the β memories could be addressed identically to the α memories. This design requires 512 MLABs and 512 M10K blocks.

Our second design solves all 256 of the *Q(x)* terms in parallel. In this design, we can no longer reuse memories, so the layouts are changed so that each memory holds half as many values as in the first design. As before, we store the β_i coefficients in MLABs. Here, each memory consists of two MLABs configured to be 16 bits wide x 64 words deep and holds the β_i coefficient for eight of the *Q(x)* equations.

The α coefficients are stored in M10K memories. Each memory stores the α for a single *Q(x)* equation and consists of four M10K blocks configured as 64 bits wide x 256 words deep. These memories are true dual-port ROMs with addresses *n* and *n+64* read simultaneously to permit the 128-bit reads necessary to support our pipeline. Half of the words are unused. As with the first design, the α and β memories share address signals, and all of the coefficients can be read in 64 cycles. This design requires 512 MLABs and 1024 M10K blocks.

Table III. The memory layout used in our first design. The subscripts on α and β indicate which of the Q(x) terms the coefficients are associated with.

	4 MLABs (16 bits wide x 128 words deep)			
addr 0	$\beta_{n}(1..0)$	$\beta_{n+1}(1..0)$	…	$\beta_{n+7}(1..0)$
⋮	⋮	⋮		⋮
addr 63	$\beta_{n}(127..126)$	$\beta_{n+1}(127..126)$	…	$\beta_{n+7}(127..126)$
addr 64	$\beta_{n+128}(1..0)$	$\beta_{n+129}(1..0)$	…	$\beta_{n+135}(1..0)$
⋮	⋮	⋮		⋮
addr 128	$\beta_{n+128}(127..126)$	$\beta_{n+129}(127..126)$	…	$\beta_{n+135}(127..126)$
addr129				
⋮				
addr 255				

4 M10K (128 bits wide x 256 words deep)
$\alpha_{n}(127..0)$
⋮
$\alpha_{n}(8191..8064)$
$\alpha_{n+128}(127..0)$
⋮
$\alpha_{n+128}(8191..8064)$
0
⋮
0

Table IV. The memory layout used in our second design. The subscripts on α and β indicate which of the Q(x) terms the coefficients are associated with.

	2 MLABs (16 bits wide x 64 words deep)				**4 M10K (64 bits wide x 256 words deep)**
addr 0	β_n(1..0)	β_{n+1}(1..0)	…	β_{n+7}(1..0)	α_n(63..0)
⋮	⋮	⋮		⋮	⋮
addr 63	β_n(127..126)	β_{n+1}(127..126)	…	β_{n+7}(127..126)	α_n(8127..8064)
addr 64					α_{n+128}(127..64)
⋮					⋮
addr 128					α_{n+128}(8191..8128)
addr129					*0*
⋮					⋮
addr 255					*0*

In Virtex 4 devices, two block RAMs can be configured in a single-port configuration 128 bits wide and 256 words deep. In our first design, half of these words could be used to store the nonlinear coefficients for two of the *Q(x)*. Alternatively, a true dual-port configuration 64 bits wide by 512 words deep could also be accommodated by two block RAMs. These configurations allow nonlinear coefficient storage in the same manner as we have implemented in the Cyclone V devices. In both devices, half of the memory words are unused. In Virtex 4 devices, the linear coefficients can be stored in block RAM or distributed memory. For our first design, four block RAMs could be used to store all of the linear coefficients. They could be arranged as a single 256 bit wide by 256 word deep memory or as two 128 bit x 256 word memories. Half of the memory locations would be unused. For the second design eight block RAMs could have been configured as a single 512 bit wide x 256 word deep memory or as two 256 bit x 256 word memories. In this case, 75% of the memories' capacities would be unused. Alternatively, each slice can be configured as a 64-bit memory. Then 32 slices could store the linear coefficients for 16 of the *Q(x)* equations in the configuration from Table III and 16 slices could store the coefficients for 8 of the *Q(x)* equations in the configuration from Table IV.

2.6. Timing Optimization

The QUAD algorithm has a regular structure that is readily parallelizable. The processing consists of simple combinatorial logic that is easily pipelined to maintain fast clock frequencies. To ensure fast clocks in parallel QUAD implementations, it is important to floorplan the design to keep the logic and memories associated with each *Q(x)* equation close together. Due to the structure of the FPGAs, the dedicated memory resources storing the nonlinear coefficients will generally be spread across the device and will dictate placement of the processing logic. Due to this and the large fanouts of some signals, such as memory addresses and the combinations of state, it is also helpful to duplicate some of these signals.

Table V. Comparison of our results to previous FPGA implementations. Note that QUAD low and QUAD medium replace S with a pseudorandom function.

	Freq. (MHz)	Area	Thru. (Mb/s)	Thru./Area
QUAD 1 (Cyclone V)	164	8723 ALUTs 2,097,152 mem. bits	157.3	18.0kbps/ALUT
QUAD 2 (Cyclone V)	143	15612 ALUTS 2,129,920 mem. bits	265.2	17.0kbps/ALUT
QUAD low (Virtex-4) [2, 7]	267	85 slices	0.016	0.2 kbps/slice
QUAD med. (Virtex-4) [2, 7]	262	406 slices	4.1	10.1 kbps/slice

2.7. Results

A comparison of our results to previously published designs is shown in Table V. Note that the previous designs, QUAD low and QUAD medium, replace the random coefficients with pseudorandom functions. This violates the security argument in [1] but greatly reduces the circuit area. Cyclone-V ALUTs and Virtex-4 slices are not equivalent structures, although they are similar. Using order of magnitude estimates, our first Cyclone-based design is about 100x larger in physical area than the QUAD low, Virtex-based design, but it has over 1000X greater throughput, measured in bits-per-second of keystream generated. It is more than 10x larger than the QUAD medium, Virtex-based design, but its throughput is about 40x higher. Our second Cyclone-based design is about 1000X larger in physical area than the QUAD low, Virtex-based design, but it has a throughput over 15,000x greater. It is also about 40x larger and 60x faster than the QUAD medium, Virtex-based design.

2.8. Increasing Throughput

The throughput can be further increased, at the cost of additional hardware, by further parallelizing the computation. As described in the section on computations of state, generating more than 128 combinations of state per cycle is straightforward. For instance, our second design could be adapted to have a 256-bit data path. This would eliminate 32 cycles from the pipeline, but would also require 256 additional linear and nonlinear processing units and modifications to the memories. The area and throughput of the design would both approximately double, although the increased logic complexity would likely reduce the achievable clock frequency and limit the throughput increase to a factor less than two.

3. Conclusion

We have demonstrated a variety of different implementations of the QUAD stream cipher algorithm, including software implementations in a PC and an ARM microprocessor, and hardware implementations on the Cyclone V FPGA. All of our implementations are over GF(2) and output 128 keystream bits per cycle from a 256-bit internal state. Our fastest implementations are over 3.3Mbits/sec on the PC, 580kbits/sec on the ARM processor, and 265Mbits/sec on the Cyclone V FPGA. We investigated design variations to improve the speed of the implementations, and we discussed methods for further improving the software and hardware approaches in future work.

References

1. C. Berbain, H. Gilbert, and J. Patarin, "QUAD: A practical stream cipher with provable security," in *Advances in Cryptology - EUROCRYPT 2006* (S. Vaudenay, ed.), vol. 4004 of *Lecture Notes in Computer Science*, pp. 109–128, Springer Berlin / Heidelberg, 2006.

2. Yang, B.-Y., Chen, O.C.-H., Bernstein, D.J., Chen, J.-M.: Analysis of QUAD. Pages 290--308 in *Fast software encryption: 14th international workshop, FSE 2007, Luxembourg, Luxembourg, March 26--28, 2007, revised selected papers*, edited by Alex Biryukov. Lecture Notes in Computer Science 4593, Springer, 2007. ISBN 978-3-540-74617-1.

3. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/ Accessed Jan. 28,2013.

4. International Organization for Standardization, ISO/IEC 18031:2011, Random bit generation. 2011. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=54945 Accessed Jan. 28,2013.

5. Morgan, T.P., "ARM Holdings Eager for PC and Server Expansion," *The Register*, Feb.1, 2011.

6. D. Arditti, C. Berbain, O. Billet, and H. Gilbert, "Compact fpga implementations of quad," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, ASIACCS '07, (New York,NY, USA), pp. 347–349, ACM, 2007.

7. D. Arditti, C. Berbain, O. Billet, H. Gilbert, and J. Patarin, "QUAD: Overview and recent developments," in *Symmetric Cryptography* (E. Biham, H. Handschuh, S. Lucks, and V. Rijmen, eds.), no. 07021 in Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Internationales Begegnungsund Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

8. Altera, Cyclone V Device Handbook, http://www.altera.com/literature/lit-cyclone-v.jsp, accessed 12/18/2012.

9. Xilinx, Virtex-4 FPGA User Guide UG070 (v2.6) Dec. 1, 2008. http://www.xilinx.com/support/documentation/user_guides/ug070.pdf Accessed 12/20/2012.