An Efficient Hardware Architecture for Spectral Hash Algorithm

Ray C.C. Cheung
Department of Electrical Engineering
University of California Los Angeles
rcheung@ee.ucla.edu

Çetin Kaya Koç
University of California Santa Barbara &
City University of Istanbul
koc@cs.ucsb.edu

John D. Villasenor
Department of Electrical Engineering
University of California Los Angeles
villa@ee.ucla.edu

Abstract

The Spectral Hash algorithm is one of the Round 1 candidates for the SHA-3 family, and is based on spectral arithmetic over a finite field, involving multidimensional discrete Fourier transformations over a finite field, data dependent permutations, Rubic-type rotations, and affine and nonlinear functions. The underlying mathematical structures and operations pose interesting and challenging tasks for computer architects and hardware designers to create fast, efficient, and compact ASIC and FPGA realizations. In this paper, we present an efficient hardware architecture for the full 512-bit hash computation using the spectral hash algorithm. We have created a pipelined implementation on a Xilinx Virtex-4 XC4VLX200-11 FPGA which yields 100 MHz and occupies 38,328 slices, generating a throughput of 51.2 Gbps. Our fully parallel implementation shows that the spectral hash algorithm is about 100 times faster than the fastest SHA-1 implementation, while requiring only about 13 times as many logic slices.

1. Introduction

In many areas of engineering and applied mathematics, spectral methods provide powerful tools for solving and analyzing problems. For instance, large to extremely large sizes of numbers can efficiently be multiplied by using discrete Fourier transform (DFT) and convolution property. Such computations are needed when computing π to millions of digits of precision, factoring composite integers and also big prime number search projects. The most (asymptotically) efficient algorithm for multiplication two large numbers is called the Schönhage-Strassen algorithm [1] which is a spectral algorithm. Spectral techniques (also named "frequency domain techniques") have recently been used in certain special applications in cryptography, including in hash function computation [2, 3], modular arithmetic for the RSA cryptosystem [4], and finite field arithmetic for elliptic curve cryptography [5, 6, 7].

Cryptographic applications require exact arithmetic and the underlying mathematical structures (groups, rings, and fields) have finitely many elements. Typically, the finite ring of integers modulo n, the finite field of p (where p is a prime) or 2^k elements are used, respectively represented as Z_n, GF(p) and GF(2^k). Furthermore, in cryptographic applications, the DFT computations are performed in a finite ring or finite field; this differs from many applications of the Fourier transformations in digital signal processing where floating-point or fixed-point arithmetic is used.

In response to recent advances in the cryptanalysis of hash functions, National Institute of Standards and Technology (NIST) started an international competition for a new hash standard, called SHA-3, which will eventually replace SHA-1 and SHA-2 family of hash functions [8]. In this paper, we present an efficient hardware architecture for the Spectral Hash algorithm [3] which is one of the Round 1 candidates. It is based on spectral arithmetic over a finite field, involving the DFT over a finite field, data dependent permutations (swaps), Rubic-type rotations, and affine and nonlinear functions. The spectral hash algorithm was particularly designed for hardware, and it is relatively slower in software compared to other hash functions. However, it is still a challenge to create a fast and efficient hardware implementation due to the size of the operands and diversity of the mathematical functions involved. The input size of the compression function (which is the heart of the hash function, iteratively applied to every chunk of data to be hashed) is 512 bits; subsequently, the spectral hash algorithm computes 512 bits of output, from which a hash

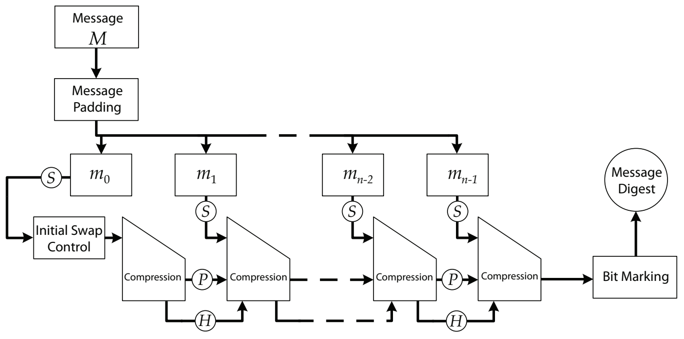

Figure 1. The augmented Merkle-Damgard scheme used in the spectral hash algorithm.

value of length that is an integer multiple of 32, from 128 up to 512, can be selected. The 512-bit input data is broken into 128 blocks, arranged as a 3-D array of dimension $4 \times 4 \times 8$ with 4-bit entries. The spectral hash algorithm makes use of 4-point and 8-point DFTs over the finite field GF(17), and also treats its 4-bit data chunks as elements of the finite field GF(2^4) generated by the primitive polynomial $p(x) = x^4 + x^3 + x^2 + x + 1$.

These mathematical structures and operations pose interesting and challenging tasks for computer architects and hardware designers to create fast, efficient, and compact ASIC and FPGA realizations of the spectral hash algorithm. Furthermore, the experience and knowledge obtained from implementing the spectral hash algorithm will be also highly valuable for creating other cryptographic, algebraic, and number-theoretic applications of spectral arithmetic.

2. Spectral Hash Algorithm

The spectral hash algorithm is based on the classical Merkle-Damgard scheme for hash generation; the augmented scheme used in the spectral hash algorithm is shown in Fig. 1. The input message is first padded to make its length an integer multiple of 512. It is then processed one block at a time using the compression function. In this paper, we present our implementation of the full hash function mapping any length data into a 512-bit hash value.

Once the initialization is completed, the spectral hash algorithm configures three prisms S, P, H which are 3-D arrays of dimension $4 \times 4 \times 8$ consisting of 4-bit or 7-bit numbers. The input data is 512 bits, and is mapped into a prism shown in Fig. 2. This prism is a three-dimensional data structure in i, j, and k directions. Given the ith block of data m_i (of length 512 bits), these prisms are updated according to the following compression function steps:

```
Inputs: $m_i$, $P$-prism, $H$-prism
Outputs: $P$-prism, $H$-prism
$P$ and $H$ were updated in previous step with $m_{i-1}$
Take chunk $m_i$ and form $S$-prism
$S$ = AffineTransform($S$)
$P$ = SwapControl1($S, P$)
$P$ = SwapControl2($S, P$)
$S$ = $k$-DFT($S$)
$P$ = SwapControl3($S, P$)
$S$ = $j$-DFT($S$)
$P$ = SwapControl4($S, P$)
$S$ = $i$-DFT($S$)
$S$ = NLST($S, P, H$)
$H$ = $S$
$P$ = PlaneRotate($P$)
```

In this paper, we describe our implementation of the full 512-bit spectral hash algorithm, involving all mathematical operations over the finite fields GF(2^4) and GF(17), the 3-dimensional DFT operations, the affine and nonlinear transformations, and data dependent permutations. The main contributions of this paper are as follows:

- we propose a new architecture for the spectral hash algorithm;
- we propose a new architecture for multidimensional DFT over a finite field;
- we review and compare hardware architectures for hash algorithms, with varying degrees of parallelism;
- we present our experimental results targeting Xilinx FPGAs to illustrate, evaluate, and compare our approach.

This paper is organized as follows. Section 3 shows the architecture for different stages of the proposed design. Section 4 evaluates the proposed architecture, provides FPGA implementation results, and discusses strategies for obtaining minimal area cores. Concluding remarks are given in Section 5.

3. Proposed Hardware Architecture

The general structure of the spectral hash algorithm is very suitable for hardware implementation. The whole design can be mapped into a 11-stage datapath. An overview of the proposed architecture is illustrated in Fig. 3. A straight arrow refers to the input to a particular stage, and a dotted arrow refers to a bypass to a stage. For instance, only the S-prism is used in the affine transformation stage. At the end of whole datapath, the content of all three prisms are updated.

3.1. Affine Transformation

The input to this stage is S-prism. For all i, j, k in the S-prism do the following. We take a block of S-prism S_{ijk}

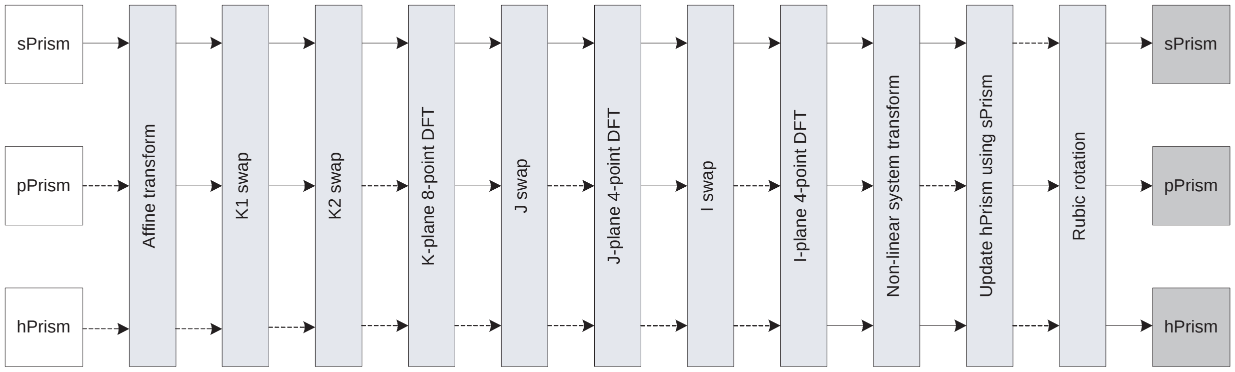

Figure 3. Overview of the proposed architecture.

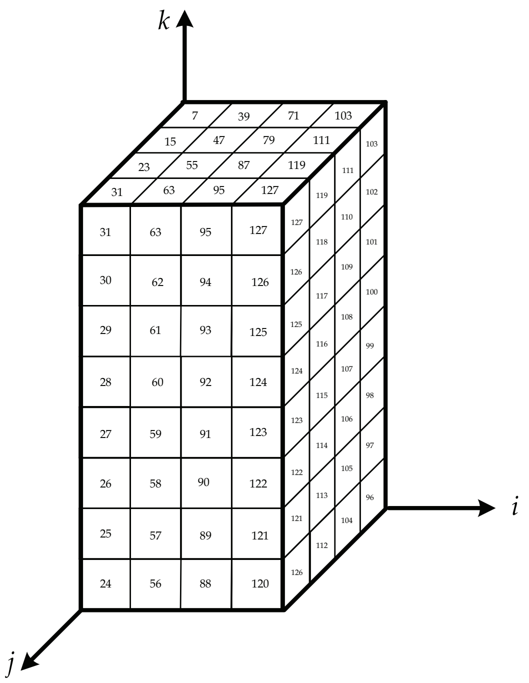

Figure 2. Example of the Prism.

and compute its inverse using

$$U = S_{ijk}^{-1} \in GF(2^4)$$

The architecture of the affine transformation is described in Fig. 4. Each block is transformed using an affine Transform Table which has 16 entries. We can iterate the transform for 128 times and use a single table lookup for the transformation. For the proposed design, each block in the S-prism is connected to an individual ROM for the table lookup. The output of the affine-ROM is a 4-bit data, thus 128 affine tables are used in this stage.

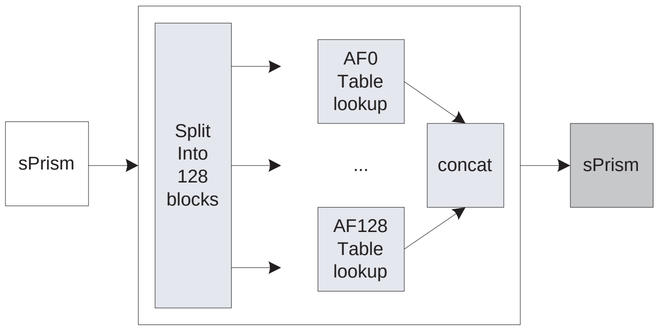

Figure 4. Architecture of affine transformation.

3.2. 3-Dimensional Simultaneous DFTs

After the affine transforms, simultaneous 3-dimensional DFTs are applied to the S-prism. The standard row-column method of DFTs is successively applied through k, j and i axes as shown in Fig. 5. The DFT used is defined over the prime field $GF(17)$. For the k-axis DFT, we have to compute 16 different 1-dimensional 8-point DFTs. For the i-axis and j-axis, we need to calculate 32 different 4-point DFTs for each axis. The equation for the DFT is as follows:

$$X_i = DFT_d(x) := \sum_{j=0}^{d-1} x_j \omega_d^{ij} \pmod{17}, \qquad (1)$$

where i = 0,1,2,. . .,$d-1$, and d is either 4 or 8. Additional technical details on the finite field DFT can be found in [3].

- k-axis DFT

  In the k-axis DFT stage, the input S-prism is indexed from block 0 to block 127. Eight consecutive blocks are computed in one computing block which eight output are produced in one clock cycle assuming there is

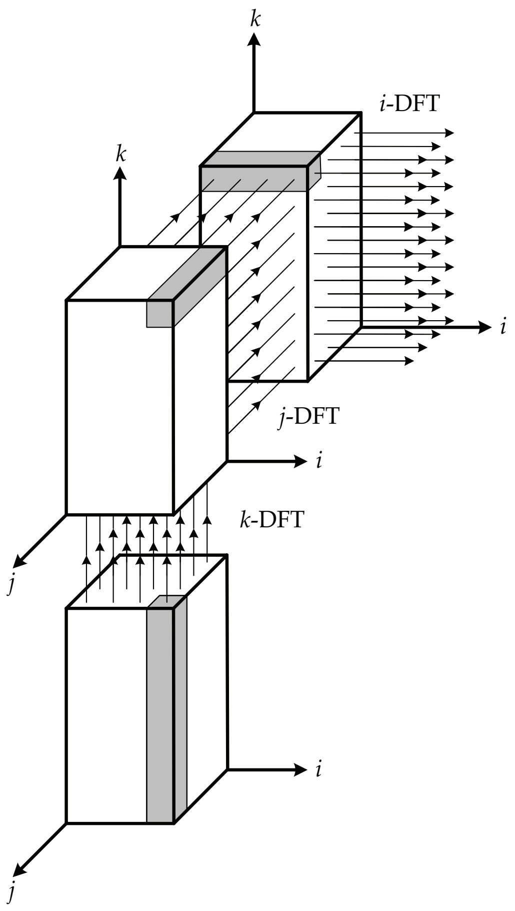

Figure 5. Example of multiple dimensional discrete Fourier transforms.

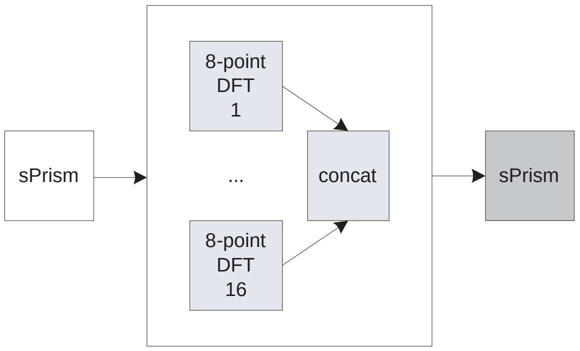

Figure 6. Architecture of k-axis DFT.

no pipeline register. As shown in Fig. 6, 16 8-point DFT blocks are used in parallel. The output blocks are concatenated and stored back to the S-prism for the next stage.

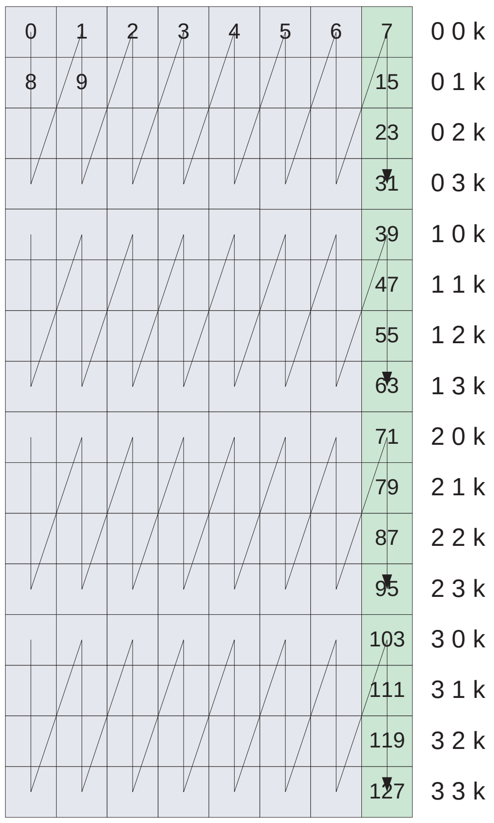

Figure 7. j-axis DFT wiring pattern. The index on the right $(00k)$ refers to $i = 0, j = 0, k = 0 \dots 7$.

- j-axis DFT

  In order to perform j-axis DFT, the data stream of the original input S-prism should be rearranged. Previously, the data is transversal from block 0 to block 127. As shown in Fig. 7, data is rearranged into a sequence of block 0, block 8, block 16, block 24 and so on. As a result, every four data blocks are feeded into the 4-point DFT block as described in Fig. 8 for computation.

- i-axis DFT

  In the i-axis DFT design stage, two main components are the wiring block and the 4-point DFT block. Similarly, the wiring block as shown in Fig. 7 is then to group the block 0, 32, 64, 96 in S-prism into a longer multiple blocks data for the 4-point DFT block. The 4-point DFT component is identical to the one we used in j-axis DFT stage.

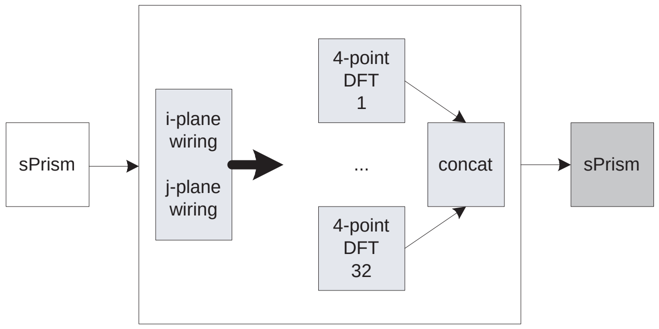

Figure 8. Architecture of i-axis & j-axis 4-point DFT.

3.3. Multi-way Swapping

Additional swapping steps for the P-prism are performed using swap control planes (sc-planes) of the S-prism after each DFT steps as shown in Fig. 9. The first sc-plane is a 4×4 array created by XORing the two planes. Each bit of the four bits in the S-prism controls one swapping, as a result, four swapping steps are performed for the 8 blocks in the k-axis. For example, the first sc-plane is generated using these explicit formulae:

$$\begin{aligned}
\text{if}\ \ S_{ij0}[0] \oplus S_{ij4}[0] &= 0 \quad \text{then} \quad \text{swap}(P_{ij0}, P_{ij7})\\
\text{if}\ \ S_{ij0}[1] \oplus S_{ij4}[1] &= 0 \quad \text{then} \quad \text{swap}(P_{ij1}, P_{ij6})\\
\text{if}\ \ S_{ij0}[2] \oplus S_{ij4}[2] &= 0 \quad \text{then} \quad \text{swap}(P_{ij2}, P_{ij5})\\
\text{if}\ \ S_{ij0}[3] \oplus S_{ij4}[3] &= 0 \quad \text{then} \quad \text{swap}(P_{ij3}, P_{ij4})
\end{aligned}$$

- k-axis Swapping

  Fig. 10 shows the top control path and the bottom data path for the swapping block. Each swap module contains two inputs, one control signal, and two outputs. For the k1-swapping step, the selection block uses item 1 and 5 while using item 2 and 6 for the k2-swapping step. These two signals from the selection block are XORed to become the swap select signal for the swapping block. Note that 128 blocks of the S-prism and P-prism are divided into 16 rows for different i and j values. The k-axis swapping are performed in parallel.

- j-axis and i-axis Swapping

  For the j-axis and i-axis swapping, control path and data path are constructed in a similar manner. The main difference for the control path is the use of different i, j, and k values. For the data path, multiple swapping steps as shown in Fig. 11 may be applied to the same P-prism block, which means the delay path is longer.

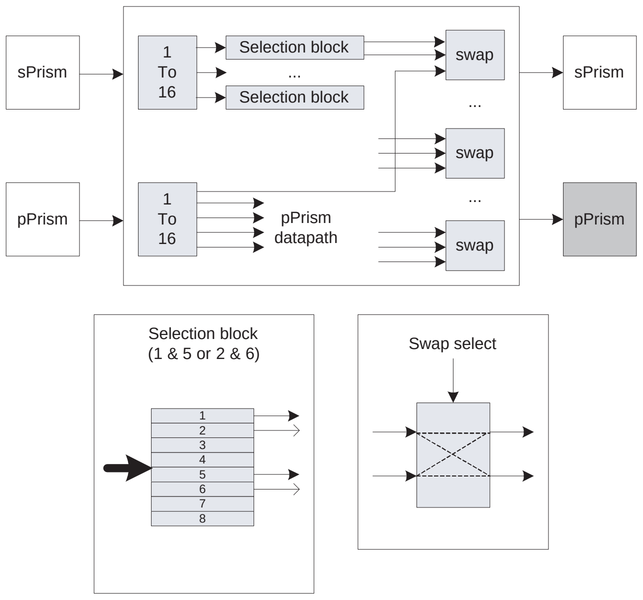

Figure 10. Architecture of the multi-way swapping.

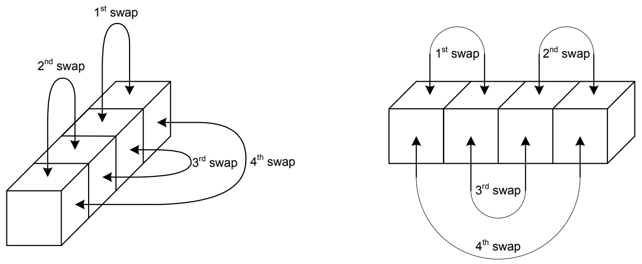

Figure 11. Example of multiple swapping on the same block.

3.4. Nonlinear System Transformation

The nonlinear system transformation (NLST) is used to combine the data from the S-prism and P-prism and store the output to the S-prism. Note that the intermediate result computed in this stage is used to index one of the data block in the input S-prism. The design component contains the most hardware resources of the whole architecture. For all i, j, k do the following on S-prism:

$$S_{ijk} = \left(S'_{ijk} \oplus PL_{ijk}\right)^{-1} \oplus \left(S'_{P_{ijk}} \oplus PH_{ijk}\right)^{-1} \oplus H_{ijk}$$

where

$$\begin{aligned}
S'_{ijk} &= S_{ijk} \pmod{16} \quad \text{for all}\ \ i, j, k\\
PL'_{ijk} &= P_{ijk} \pmod{16} \quad \text{for all}\ \ i, j, k
\end{aligned}$$

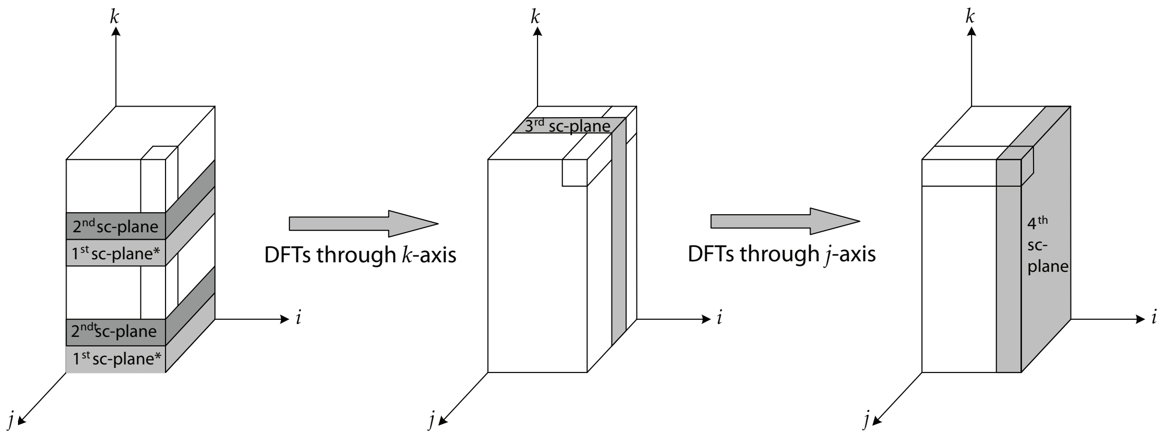

Figure 9. Example of swapping planes.

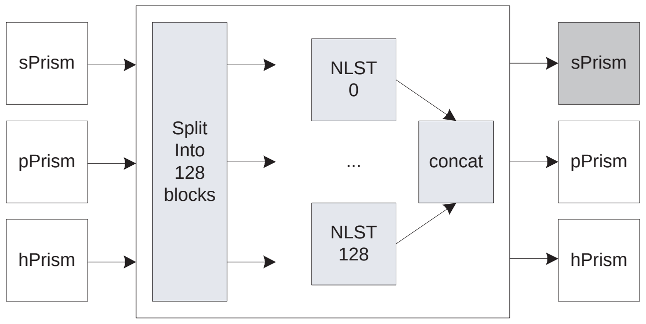

Figure 12. Parallel NLST architecture.

$$PH'_{ijk} \quad = \quad (S_{ijk} \ \text{div} \ 16) \,||\, (P_{ijk} \ \text{div} \ 16) \quad \text{for all} \ \ i, j, k$$

Note that GF(17) produces 5-bit numbers, in the range $[0, 16]$. With the above nonlinear transformations, they are reduced back to 4 bits. For the proposed architecture as shown in Fig. 12, each NLIST_i block uses two inverse table lookups, and a three-stage MUX for selecting the input S-prism. In order to produce a fully parallel design, this block is replicated by 128 times. For the 3-stage MUX as shown in Fig. 13, the three selection signals are controlled by the 7 bits from one of the block in the P-prism that has been bit selected into 3 segments, 2-bit for i, 2-bit for j and 3-bit for k. This 3-stage MUX produces a 4-bit element for the NLST computation component.

3.5. Rubic Rotation

The P-prism performs some special rotations according to the k values, the kth plane uses kth Rubic rotations as shown in Fig. 14. We can describe the plane rotations along

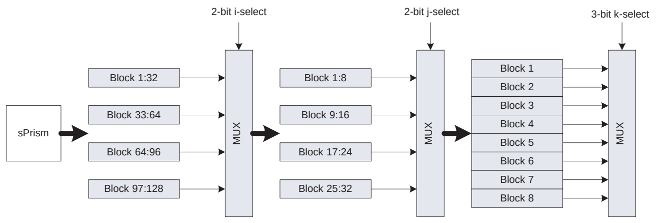

Figure 13. 3-stage MUX for the block selection.

the k-axis as follows:

$$\begin{array}{llll}
\text{if} & (k = 0 \bmod 4) & \text{then} & P_{ijk} = P_{ijk} \\
\text{if} & (k = 1 \bmod 4) & \text{then} & P_{ijk} = P_{(3-j)ik} \\
\text{if} & (k = 2 \bmod 4) & \text{then} & P_{ijk} = P_{jik} \\
\text{if} & (k = 3 \bmod 4) & \text{then} & P_{ijk} = P_{j(3-i)k}
\end{array}$$

For the hardware architecture, it is basically simple wiring patterns as shown in Fig. 15. For the P-prism is divided into 8 parts that undergo 4 types of wirings. One of the ROT3 wiring is described in this paper.

4. Experimental Results

The FPGA implementations presented in this section use Xilinx System Generator 9.1.01i to generate VHDL designs. The generated designs are mapped on a Xilinx Virtex-4 XC4VLX200-11 device. Xilinx XST 9.1.03i is used for synthesis and Xilinx ISE 9.1.03i is used for placement and routing. The area comparison for different stages of the datapath is shown in Table 4. Z^{-1} refers to adding one additional pipeline stage into the datapath for improving the overall data throughput. Since each slice in FPGA

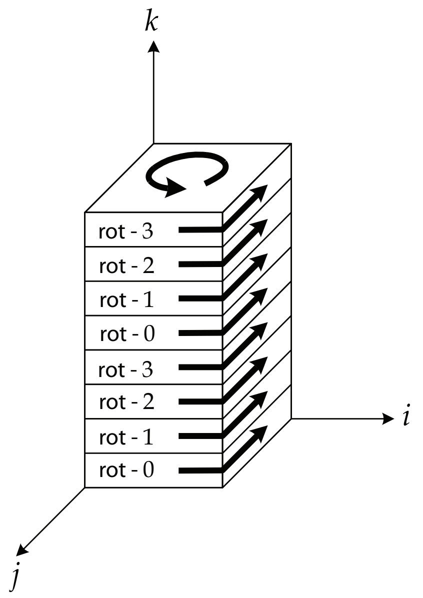

Figure 14. Example of Rubic rotations.

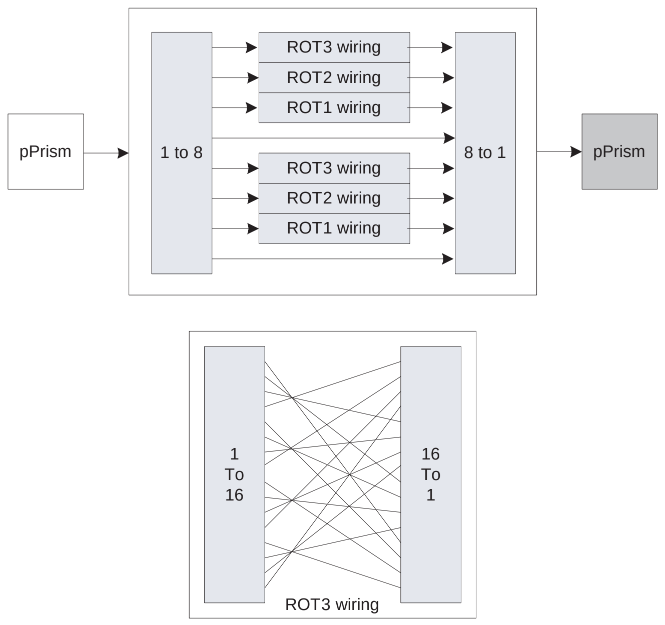

Figure 15. Architecture of Rubic rotations.

architecture provides lookup tables and register elements, for our design, adding extra pipeline stage do not consume extra slices but shorten the critical delay path, thus further improve the throughput of the design. In the table, memory slices refer to SLICEM block whose lookup table can be configured as memory or shift register SRL16, while slices refer to both SLICEM and SLICEL blocks whose lookup table can only be used as logic. For instance in the

Table 2. Memory bits usage comparison.

Component	Memory bits
affine transform	8 kbits
k-plane 8-pt DFT	200 kbits
j-plane 4-pt DFT	200 kbits
i-plane 4-pt DFT	200 kbits
NLST	16 kbits

Table 3. Related work comparison.

Design	Throughput	Area (slices)	Clock
MCEvoy [9] SHA2-256	1,009 Mbps	1,373 Virtex2	133 MHz
MCEvoy [9] SHA2-512	1,466 Mbps	1,466 Virtex2	66 MHz
McLoone [10] SHA1-512	479 Mbps	2,914 VirtexE	38 MHz
Proposed SHA3-512	51,200 Mbps	38,328 Virtex4	100 MHz

NLST stage, the 8,192 SLICEM out of 17,920 total slices are mainly used for the 3-stage MUX component. We can see that the compression engine is able to run at a maximum 100 MHz for the proposed architecture. Note that for this design, all the logic are placed into slice-only mode, so no DSP block or Block RAM is used. For the fully parallelized design, one computed S-prism is produced every clock cycle with an initial latency of 11 cycles for the un-pipelined design, whereas it takes 15 cycles for the pipelined design.

The memory bits usage are presented in Table 4. The memory bits for i-/ j-/ k-DFT come from the modular table for the prime field, while from the inverse-table for the NLST and affine-table for the affine transformation. Several related publications have proposed various hardware optimization techniques for the SHA-1 and SHA-2 hash family as shown in Table 4. For example, the most efficient SHA2-256 design is able to run at over 133 MHz and produces a throughput at over 1,000 Mbps. Another previous publication has reported a SHA1 design running at 38 MHz and producing a data throughput of 479 Mbps. Since there is yet no existing SHA-3 hardware design reported in literature, in this paper we do not provide a direct comparison with other SHA-3 algorithms. We will compare our design to them as they become available.

If the area usage has higher priority than the timing concerns for particular embedded designs, our proposed architecture can meet such requirements by using multiple-cycle components for each design stage. For example, the affine transformation step can be reduced into one single component for 128 iterations, using a rotation component. When the table lookup is completed, and $DONE$ signal will be sent to the next design stage as an $ENABLE$ signal. The key advantage of the proposed approach is its flexibility, which gives the designer the option of obtaining higher

Table 1. Area usage using a Xilinx Virtex-4 LX200-11FF1513.

Component	Area (slices)	Area (memory slices)	Delay
affine transform	256	0	6.676ns
$k1$-swap	952	0	9.935ns
$k2$-swap	616	0	9.705ns
k-axis 8-pt DFT	7,424	2,400	14.889ns
k-axis 8-pt DFT (Z^{-1})	7,424	2,400	9.631ns
j-swap	1,036	0	9.951ns
j-axis 4-pt DFT	4,288	640	13.446ns
j-axis 4-pt DFT (Z^{-1})	4,288	640	9.631ns
i-swap	1,036	0	9.951ns
i-axis 4-pt DFT	4,288	640	12.127ns
i-axis 4-pt DFT (Z^{-1})	4,288	640	8.932ns
NLST	17,920	8,192	14.094ns
NLST (Z^{-1})	17,920	8,192	9.983ns
ROT	512	0	5.710ns
Total	38,328	11,872	9.983ns

throughput at the cost of increased area. In fact the proposed design uses about 43 % of the Virtex-4 LX200 FPGA, and can be mapped into the Virtex-4 LX100 FPGA. One of the limitation of the current design is the insufficient number of I/O pins for the FPGA chip in order to achieve very high data throughput, however it can be readily surmounted when the design is realized in an ASIC design flow.

5. Conclusions

We have presented a fully parallelized architecture for the spectral hash algorithm. The pipelined design for the compression engine is running at 100 MHz and provides a data throughput at 51.2 Gbits per second for computing a full 512-bit hash value. Current and future work includes extending the design to the overall hashing, further optimizing the proposed design throughput, and exploring the tradeoffs between a fully parallelized design and an area-efficient flexible architecture. For example, it is valuable to calculate the right amount of hardware usage for each stage that would give the best area-time hardware for the spectral hash algorithm. We are also interested in comparing our hardware implementation to other SHA-3 candidate algorithms as their hardware implementations become available.

Furthermore, we believe the experience and knowledge obtained from implementing the spectral hash algorithm in hardware is highly valuable for creating other hardware designs using spectral arithmetic, such as RSA and elliptic curve cryptography [4, 5, 6, 7].

References

[1] A. Schönhage and V. Strassen, “Schnelle multiplikation grosser zahlen,” *Computing*, vol. 7, pp. 281–292, 1971.

[2] C. P. Schnorr, “FFT-Hashing: An efficient cryptographic hash function,” in *Rump Session of Crypto 91*. Springer, Aug. 1991.

[3] G. Saldamlı, C. Demirkıran, M. Maguire, C. Minden, J. Topper, A. Troesch, C. Walker, and Ç. K. Koç, “Spectral hash,” in *The First SHA-3 Candidate Conference*. Katholieke Universiteit, Leuven, Belgium, February 25-28, 2009.

[4] G. Saldamlı and Ç. K. Koç, “Spectral modular exponentiation,” in *Proceedings, 18th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Montpellier, France, June 25-27 2007, pp. 123–130.

[5] G. Saldamlı, “Spectral Modular Arithmetic,” Ph.D. dissertation, Oregon State University, May 2005.

[6] S. Baktır and B. Sunar, “Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography,” in *Proceedings of the 21st International Symposium on Computer and Information Sciences (ISCIS 2006)*. LNCS, Volume 4263, Springer, Oct. 2006, pp. 991–1001.

[7] ——, “Achieving efficient polynomial multiplication in Fermat fields using the fast Fourier transform,” in *Proceedings of the 44th ACM Southeast Conference (ACMSE 2006)*. ACM Press, Mar. 2006, pp. 549–554.

[8] NIST, “Cryptographic Hash Algorithm Competition,” 2009, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[9] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, “Optimisation of the SHA-2 family of hash functions on FPGAs,” in *ISVLSI’06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, Mar. 2006, pp. 317–322.

[10] M. McLoone and J. V. McCanny, “Efficient single-chip implementation of SHA-384 & SHA-512,” in *FPT’02: Proceedings of the IEEE International Conference on Field Programmable Technology*, Dec. 2002, pp. 311–314.