FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash

Brian Baldwin, Andrew Byrne, Mark Hamilton, Neil Hanley,
Robert P. McEvoy, Weibo Pan and William P. Marnane
Claude Shannon Institute for Discrete Mathematics, Coding and Cryptography.
Department of Electrical & Electronic Engineering,
University College Cork, Cork, IRELAND
Email: {brianb, andrewb, markh, neilh, robertmce, weibop, liam}@eleceng.ucc.ie

***Abstract*—Hash functions are widely used in, and form an important part of many cryptographic protocols. Currently, a public competition is underway to find a new hash algorithm(s) for inclusion in the NIST Secure Hash Standard (SHA-3). Computational efficiency of the algorithms in hardware will form one of the evaluation criteria. In this paper, we focus on five of these candidate algorithms, namely CubeHash, Grøstl, LANE, Shabal and Spectral Hash. Using Xilinx Spartan-3 and Virtex-5 FPGAs, we present architectures for each of these hash functions, and explore area-speed trade-offs in each design. The efficiency of various architectures for the five hash functions is compared in terms of throughput per unit area. To the best of the authors' knowledge, this is the first such comparison of these SHA-3 candidates in the literature.**

I. INTRODUCTION

Hash functions operate at the heart of contemporary cryptographic protocols, such as the Digital Signature Standard (DSS), Transport Layer Security (TLS), Internet Protocol Security (IPSec), random number generation algorithms, authentication algorithms and password storage mechanisms.

A hash function $\mathcal{H}$ maps a message x of variable length to a string of fixed length. The process of applying $\mathcal{H}$ to x is called 'hashing', and the output $\mathcal{H}(x)$ is called the 'message hash' or 'message digest'. Cryptographic hash functions are hash functions that possess specific properties such as one-wayness [1], which means that for a given hash value y, it should be computationally infeasible to find an input x such that $\mathcal{H}(x) = y$. Another important property for cryptographic hash functions is collision resistance, which means that it should be computationally infeasible to find any two distinct inputs x_1 and x_2, such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$.

The SHA-1 (Secure Hash Algorithm) function, introduced by NIST in 1995, produces message hashes of length 160 bits. In 2002, NIST published three new hash functions with longer hash lengths: SHA-256, SHA-384 and SHA-512. In 2004, SHA-224 was added to the standard, and these four algorithms form the SHA-2 family of hash functions [2]. Due to security concerns [3], the trend in the cryptographic community is to move away from using older hash functions like SHA-1, towards newer functions like those in the SHA-2 family [4].

Currently, NIST is holding a public competition to develop a new cryptographic hash algorithm(s) [5], similar to the contest held to choose the Advanced Encryption Standard (AES) algorithm [6]. The new hash function(s) will be called SHA-3 (or the Advanced Hash Standard (AHS)), and may ultimately supersede the functions in the SHA-2 family. The contest has received 64 submissions from designers all around the world, and 51 of these designs recently progressed through to the first round of the contest. These hash algorithms are available for public comment and scrutiny, and such research is vital to the selection process. In particular, NIST has stated that computational efficiency of the algorithms in hardware, over a wide range of platforms, will be addressed during the second round of the contest [7].

In this paper we present FPGA hardware implementations of five of the SHA-3 candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. FPGAs are an attractive choice for implementing cryptographic algorithms, because of their low cost relative to ASICs. FPGAs are flexible when adopting security protocol upgrades, as they can be re-programmed in-place, and FPGAs also allow rapid prototyping of designs. Our goal is to explore area-speed trade-offs in the implementations of the five hash functions, and to compare the efficiency of the designs by examining the throughput per unit area metric.

The rest of this paper is organised as follows. Section II gives an overview of the hash function architectures, and describes the wrapper used in the evaluation of the designs. Sections III–VII respectively describe the FPGA implementations of the five hash functions in this case study. For each hash function, its specification is briefly described; an exploration of the design space is presented; and implementation results on the Spartan-3 and Virtex-5 FPGA platforms are supplied. Section VIII concludes the paper by comparing the results of the various designs.

II. OVERVIEW OF THE HASH FUNCTION ARCHITECTURES

In the design of the hash function architectures described in this paper, our main goal was to optimise throughput. Throughput is calculated as follows:

$$\text{Throughput} = \frac{\text{\# Bits in a message block} \times \text{Maximum clock frequency}}{\text{\# Clock cycles per message block}}$$

High-throughput hash function implementations are beneficial, for example in network server applications. A secondary goal was to analyse the throughput per slice of the architectures, to determine which hash function implementations make

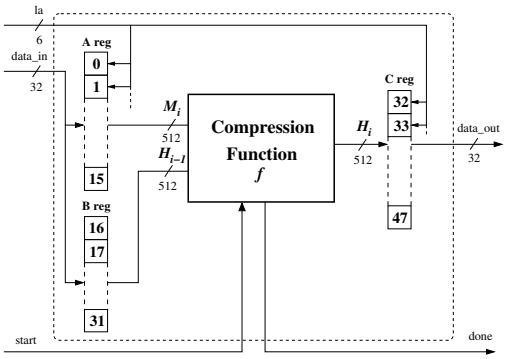

Fig. 1. Wrapper used to obtain Place & Route results

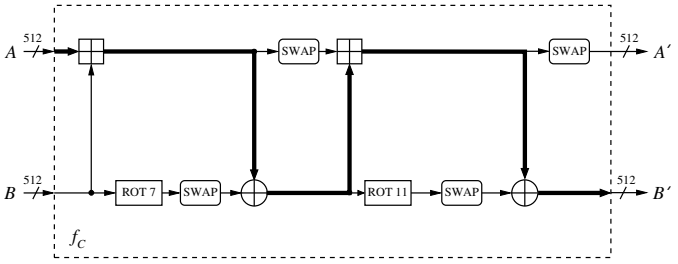

Fig. 2. Block diagram of the CubeHash compression function

the most efficient use of FPGA area. All of the hash algorithms examined in this paper include initialisation, padding and finalisation stages. However, we focused on implementing the compression function f of each hash function. The compression functions perform the majority of the computations in the hash algorithms, and the throughput of the algorithms is largely determined by the throughput of the compression functions (this is especially true for large message sizes). A webpage called the 'SHA-3 Zoo' has been set up by the Institute for Applied Information Processing and Communications (IAIK) in Graz, to track hardware implementation results of SHA-3 candidates [8]. In the language of the authors of the SHA-3 Zoo, the architectures described in this paper can be described as 'core functionality' implementations.

To allow direct substitution for the functions in the SHA-2 family, it is a requirement of the contest that the SHA-3 algorithms have output lengths $n \in \{224, 256, 384, 512\}$. For the LANE and Grøstl hash functions examined in this paper, we designed two architectures (per hash function) in order to produce all of the required message digest lengths. For each of the three other hash functions in the study, a single architecture can produce all of the required message digest lengths. Within each of these seven architectures, various area-speed trade-offs were also investigated.

Two FPGA platforms were targeted in the study: the low-cost Xilinx Spartan-3 *xc3s5000-5fg900*, and the newer technology Xilinx Virtex-5 *xc5vlx220-2-ff1760*. Results for a particular hash function architecture on the two FPGA platforms cannot be directly compared, since these two platforms have different underlying technologies. Of course, comparing different architectures on the same FPGA platform is a fair comparison. Each compression function design f was captured using VHDL, and Synthesis, Place and Route were carried out using Xilinx ISE v9.1i. We measure the area of compression function designs in FPGA slices, as given in the Map reports.

Each compression function f has a very large number of input and output signals; therefore, the Place and Route process will not be able to proceed past Mapping without including a wrapper (similar to [9]). Our wrapper, illustrated in Fig. 1, consists of registers and multiplexers, and is used to connect the I/O buses of f to 32-bit I/O interfaces at the FPGA pads. The registers also ensure that the wrapper does not affect the critical path of the compression function f. Synthesis, Place and Route are carried out on the wrapped compression functions, and the maximum allowable clock frequency is obtained from the Post-Place and Route static timing report.

The following sections detail the implementations of the five hash functions. Note that in each section, the notation used to describe the operations of each hash function is local to that section.

III. CUBEHASH

CubeHash was submitted by Bernstein to the SHA-3 contest [10]. CubeHash is defined by three parameters:

- $b \in \{1, 2, 3, \ldots, 128\}$, the number of bytes in a block of the padded message;
- $r \in \{1, 2, 3, \ldots\}$, the number of times the compression function is iterated for each padded message block;
- $h \in \{8, 16, 24, \ldots, 512\}$, the number of bits in the message digest.

Parameters r and b allow security/performance tradeoffs in different CubeHash implementations, and a particular version of the algorithm is then specified as CubeHash$r$/b-h.

A. Specification

All of the CubeHash variants use a 1024-bit state, that is represented as thirty-two 32-bit words $x[t]$, $0 \leq t \leq 31$, where each word is interpreted in little-endian form. During the algorithm's execution, the state is operated upon by a compression function, denoted here by f_C and described below. CubeHash$r$/b-h uses the following operations to compute the hash of a message M:

- Initialisation. State words $x[0]$, $x[1]$ and $x[2]$ are respectively set to integers $\frac{h}{8}$, b and r, and the remaining 29 state words are set to zero. The round function f_C is then applied to the state $10r$ times.
- Message Padding and Parsing. The message M is padded and parsed into N b-byte blocks M_i, $0 \leq i \leq N-1$.
- Message Compression using f_C. Block M_i is XORed into the first b bytes of the state, and the round function f_C is applied to the state r times. This sequence of XORing and applying f_C is performed on each of the padded message blocks.
- Finalisation and Output. After block M_{N-1} has been processed, the integer 1 is XORed with the last state word, and the round function is applied to the state a further $10r$ times. The message digest comprises the first h bits of the state.

A block diagram of the CubeHash compression function f_C is shown in Fig. 2. Essentially, the compression function has two 512-bit inputs, labelled A and B, which are formed by splitting the 1024-bit state in half. The two 512-bit datapaths then undergo a series of operations, comprising:

- 2×16 additions modulo 2^{32} (denoted by $\boxplus$), where the datapath A is added word-by-word to datapath B;
- 2×16 32-bit Boolean XORs (denoted by $\oplus$), where the two datapaths are XORed word-by-word;
- 2×16 rotation operations, where each word in datapath B is cyclically rotated upwards by a fixed number of bits;
- 4×8 swapping operations, where specified words in a datapath exchange positions.

If required, the outputs of the compression function, labelled A' and B', can be fed directly to the inputs A and B of the next iteration of the compression function.

B. Compression Function Implementation & Results

In this paper, we designed FPGA implementations of the CubeHash compression function with parameters $r = 8$ and $b = 1$, as recommended by Bernstein [10]. The rotation and swapping operations can be implemented in hardware by simply re-labelling the relevant signals. Since the state comprises 1024 bits, the same architecture for f_C can be used to produce message digests with any of the lengths required for SHA-3 (i.e. $h \in \{224, 256, 384, \ldots, 512\}$). Therefore, a CubeHash8/1-256 implementation will have the same throughput and throughput per slice performance as a CubeHash8/1-512 implementation. Similarly, the implementations of Shabal and Spectral Hash described in Sections VI and VII respectively use a single architecture to produce all of the required message digest lengths.

The critical path through the compression function consists of two modulo 2^{32} additions and two XOR operations, as indicated by the heavy lines in Fig. 2. The compression function is used $r = 8$ times for each message block M_i (i.e. for each message byte in this case, since $b = 1$). Therefore, we investigated CubeHash architectures where f_C is unrolled by various degrees. The lowest area design iteratively uses a single f_C unit and takes 8 clock cycles to process a single message block, and the highest area design uses a chain of four f_C units in series to process a single message block in two clock cycles. The results are shown in Table I. Note that the figures quoted for each design include the initial XOR of the message block with the state, and also include the area of the output register that stores the result of the last f_C calculation in the chain.

As expected, the critical path of each design increases with the degree of unrolling. However, in the 2x-unrolled Virtex-5 design, the increase in the critical path is greater than the corresponding decrease in the number of clock cycles, so an overall increase in throughput (TP) is not obtained. The larger (4x-unrolled) designs were dense and congested on the chosen FPGAs, and could not be fully routed by the routing tool. In any case, it is interesting to note that the throughput on both FPGAs is quite low, because each message block in CubeHash8/1 consists of only 1 byte.

TABLE I
CUBEHASH8/1 COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	Architecture	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3						
#1	Iterative	2883	59.26	8	0.05	0.02
#2	2x-unrolled	3268	37.90	4	0.07	**0.02**
#3	4x-unrolled	Congested Design				
Virtex-5						
#1	Iterative	1178	166.80	8	**0.16**	**0.14**
#2	2x-unrolled	1440	55.14	4	0.11	0.07
#3	4x-unrolled	Congested Design				

IV. GRØSTL

Grøstl was submitted to the SHA-3 contest by Gauravaram et al. [11]. Grøstl is an iterated hash function with a compression function built from two fixed, large distinct permutations. The design of Grøstl borrows heavily from components used in the AES block cipher, resulting in strong confusion and diffusion properties [6]. Grøstl is defined for a number of variants, denoted Grøstl-n, which return message digests of bit length $n \in \{8, 16, \ldots, 512\}$. Four variants of Grøstl are explicitly defined for the SHA-3 contest, where $n \in \{224, 256, 384, 512\}$.

A. Specification

The Grøstl variants can be divided into two categories, based on the size of the algorithm's internal state, denoted by ℓ. For Grøstl-224/256 $\ell = 512$, whereas for Grøstl-384/512 $\ell = 1024$. In this paper, we denote the Grøstl compression function by f_G.

Grøstl-n uses the following operations to compute the hash of a message M:

- Initialisation. An initial hash value H_{-1} is set to the ℓ-bit representation of n.
- Message Padding and Parsing. The message M is padded and parsed into N ℓ-bit blocks M_i, $0 \leq i \leq N-1$.
- Message Compression using f_G. Each message block M_i is combined with the previous hash value H_{i-1}, and a permutation P is applied to the result. A second permutation Q operates in parallel on M_i. The compression function output H_i is formed by XOR-ing H_{i-1} with the outputs of P and Q, as illustrated in Fig. 3.
- Output Transformation. The function $P(x) \oplus x$ is applied, where x is the final hash value H_{N-1}, and the ℓ-bit result is truncated to leave the rightmost n bits.

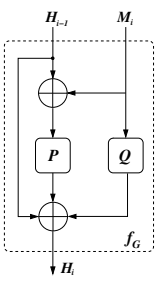

Fig. 3. Grøstl Compression function

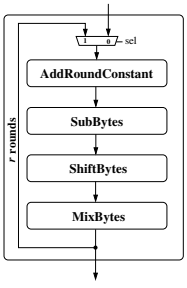

Fig. 4. Grøstl P/Q permutation

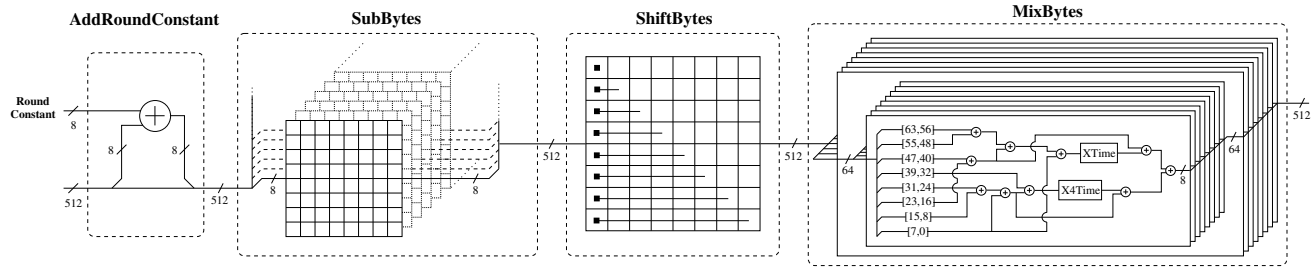

Fig. 5. Grøstl-224/256 Permutation Function Implementation

Permutations P and Q are based on the AES block cipher [6], and are illustrated in Fig. 4. Each input to P/Q is represented as a sequence of bytes; mapped to the state matrix in a similar way to the AES. When $\ell = 512$ an 8×8 matrix of bytes is used, and when $\ell = 1024$ an 8×16 matrix. Permutations P and Q are executed iteratively for r rounds. For Grøstl-224/256, the designers recommend using $r = 10$, and $r = 14$ is recommended for Grøstl-384/512. P and Q each have four round transformations, executed in the following order: (i) AddRoundConstant, where a single byte is added to the state; (ii) SubBytes, where the AES S-box is applied to each byte of the state; (iii) ShiftBytes, where each row of the state is cyclically shifted; and (iv) MixBytes, where each column of the state is transformed using matrix multiplication.

B. Compression Function Implementation & Results

Permutations P and Q are identical except for the execution of the AddRoundConstant step, where different round constants are used. Therefore, when implementing f_G, one design choice is to compute Q in parallel by replicating the hardware for P. Another approach is to use an interleaved design, which reuses the hardware for P to compute Q, resulting in extra clock cycles but lower area.

The architecture developed for the implementation of the Grøstl permutation functions is illustrated in Fig. 5, for $\ell = 512$ (similar hardware was developed for the $\ell = 1024$ case). The first stage in each permutation is the AddRoundConstant block which simply performs an XOR on one byte of the ℓ-bit input state. The round constants are stored in distributed memory on the FPGA.

The SubBytes stage transforms the state, byte by byte, using the AES S-box. We implemented the S-boxes as look-up tables, and investigated two design choices: storage of the S-boxes in distributed memory (FPGA slice LUTs), and storage in dedicated Block RAM (BRAM). Both designs are fully parallel, allowing all bytes of the state to be transformed with minimum latency. The BRAMs are synchronous; therefore, the SubBytes transformation takes a single clock cycle to compute if implemented using BRAM.

The SwapBytes transformation was realised in hardware by simply re-labelling the bytes of the state. MixBytes is the final stage of the permutation function, and processes each column of the state matrix separately and in parallel using combinational logic. Fig. 5 illustrates the logic for transforming a single byte of a particular column. The bytes in each column are combined using a series of XOR and multiplication operations. All of these MixBytes operations were implemented in combinational logic on the FPGA. In the cases where the SubBytes S-boxes were implemented using distributed ROM, an output register (not shown in Fig. 5) was used to store the state at the output of the MixBytes transformation. This round output register is not required if the S-boxes are implemented in synchronous BRAM.

TABLE II
GRØSTL-224/256 COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	S-box	*P/Q*	Area (slices)	Max. Freq. (MHz)	# Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3							
#4	BRAM	Interleaved	3183	91.02	20	2.33	0.73
#5		Parallel	4827	71.53	10	**3.66**	**0.75**
#6	Slice LUTs	Interleaved	5508	60.37	20	1.54	0.28
#7		Parallel	8470	50.06	10	2.56	0.30
Virtex-5							
#4	BRAM	Interleaved	3184	250.43	20	6.41	**2.01**
#5		Parallel	4516	142.87	10	**7.31**	1.61
#6	Slice LUTs	Interleaved	5878	128.38	20	3.28	0.55
#7		Parallel	8196	101.89	10	5.21	0.63

TABLE III
GRØSTL-384/512 COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	S-box	*P/Q*	Area (slices)	Max. Freq. (MHz)	# Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3							
#8	BRAM	Interleaved	6313	79.61	28	2.91	**0.46**
#9		Parallel	-	-	-	-	-
#10	Slice LUTs	Interleaved	10293	50.12	28	1.83	0.17
#11		Parallel	17452	43.49	14	**3.18**	0.18
Virtex-5							
#8	BRAM	Interleaved	6368	144.03	28	5.26	**0.82**
#9		Parallel	-	-	-	-	-
#10	Slice LUTs	Interleaved	10848	111.13	28	4.06	0.37
#11		Parallel	19161	83.33	14	**6.09**	0.31

The compression function f_G for the Grøstl implementation consists of two such permutation functions, P and Q. Two XOR arrays are required to complete the compression function for the input to P, and for the final output H_i. Note that in the interleaved f_G design that uses distributed memory for the S-boxes, an additional register was placed between the S-Box and SwapBytes stages, to allow P and Q to be processed using the same hardware blocks.

The Post-Place & Route results for the f_G implementations are given in Tables II and III, for Grøstl-224/256 and Grøstl-384/512 respectively. Note that for Grøstl-384/512, the BRAM requirements of the parallel architectures exceeds the resources available on the FPGAs. Implementing the S-boxes using BRAM, available on both FPGAs, improves the clock frequency significantly while also reducing the number of slices required. This returns a higher throughput/area metric for both Grøstl variants.

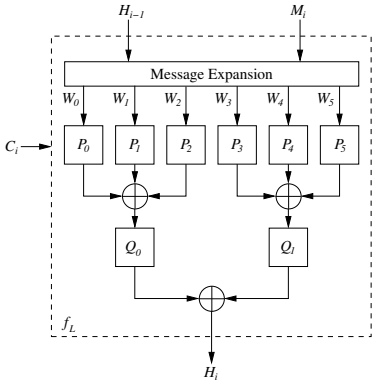

Fig. 6. LANE compression function structure

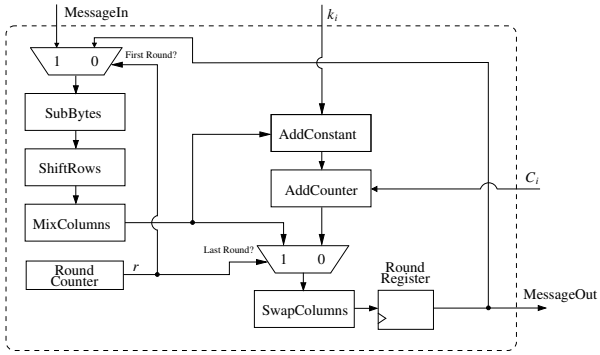

Fig. 7. Block diagram of the P_i/Q_j permutation

V. LANE

The LANE hash function, developed by Indesteege et al. is another candidate for the SHA-3 competition [12]. It is an iterated hash function with four variants, LANE-224, LANE-256, LANE-384, and LANE-512, where the suffix denotes the message digest size in bits.

A. Specification

Like Grøstl, components of the AES block cipher are also used in the LANE compression function [6]. In this paper, we denote the compression function by f_L. The LANE variants can be divided into two categories, according to the size of the internal states. LANE-224 and LANE-256 use a 256-bit state (corresponding to two AES states), while LANE-384 and LANE-512 use a 512-bit state (corresponding to four AES states). Here, LANE-224/256 is described. LANE-224/256 uses the following operations to compute the hash of a message M:

- Initialisation. An initial hash value H_{-1} is computed by applying the compression function f_L to a sequence of bits, which consists of a pre-determined string, and may comprise an optional random salt.
- Message Padding and Parsing. The message M is zero-padded, and parsed into N 512-bit blocks M_i.
- Message Compression using f_L. Each message block M_i is combined with the previous hash value H_{i-1}, and a series of AES-based round transformations is applied. A counter C_i is also maintained, which tracks the number of message bits processed.
- Finalisation and Output. The hash value H_{N-1} is passed once more through the compression function, with the message input set to a string containing the message length and, if applicable, the salt value. The 256-bit result is taken as the LANE-256 output, or truncated to give the LANE-224 output.

A block diagram of the LANE compression function f_L is given in Fig. 6. The function begins with a message expansion, where M_i is combined with H_{i-1} using a series of XOR and concatenation operations. The result of the message expansion stage is six 256-bit expanded message words $W_0, \ldots, W_5$. The remainder of f_C consists of five 256-bit XOR operations and eight permutation 'lanes', labelled P_i, $i \in \{0, \ldots, 5\}$, and Q_j, $j \in \{0, 1\}$, and arranged in two layers. Each permutation is executed r times. In LANE-224/256, $r = 6$ for the P_i permutations and $r = 3$ for the Q_i permutations.

Permutation blocks P_i and Q_j are identical $\forall$ (i, j), and comprise the operations shown in Fig. 7. The SubBytes, ShiftRows and MixColumns transformations are reused from the AES block cipher, and are applied independently to each AES sub-state within the P_i or Q_j state. The AddConstant transformation causes a pre-defined round constant k_i to be XORed with part of the P_i/Q_j state. Similarly, the AddCounter transformation XORs part of the counter C_i with the state. Finally, the SwapColumns transformation swaps columns between the AES sub-states that make up the P_i/Q_j state. The AddConstant and AddCounter operations are omitted during the last round of the permutations.

LANE-384/512 has a similar structure to LANE-256, where the main differences are: (i) the message blocks are 1024-bit; (ii) the internal P_i/Q_j states are 512-bit; and (iii) $r = 8$ for the P_i permutations and $r = 4$ for the Q_j permutations.

B. Compression Function Implementation & Results

In order to implement the compression function f_L, we firstly designed hardware for the permutations P_i/Q_j in LANE-224/256 and LANE-384/512. The SubBytes function was realised using look-up tables for each AES S-box, implemented in distributed memory. The MixColumns transformation was realised using a combination of XOR and doubling operations [13]. The AddConstant and AddCounter operations simply equate to XORs. The ShiftRows and SwapColumns transformations were implemented by re-labelling the relevant signals. A counter was also required to track the round number. A register was used to store the output state between iterations of the permutation. Therefore, the permutation circuitry logic is fully combinational, from input to the output of the SwapColumns transformation.

Implementing the compression function f_L presents a trade-off, due to the parallelism in the algorithm's design. A high-area implementation of f_L uses six permutation circuits to process each W_i in parallel. Two of these circuits can subsequently be re-used to calculate Q_1 and Q_2. A low-area implementation of f_L uses just one permutation circuit to to compute P_i and Q_j $\forall$ (i, j), but requires extra control circuitry and extra clock cycles. We investigated this trade-off by implementing several architectures for f_L, with various

TABLE IV
LANE-224/256 COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	# Parallel P_i units	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3						
#12	1	5725	50.2	49	0.52	**0.09**
#13	2	8756	42.0	28	0.76	0.08
#14	6	19692	31.3	14	**1.14**	0.05
Virtex-5						
#12	1	3442	133	49	1.38	0.40
#13	2	4515	120	28	2.19	0.48
#14	6	6888	123	14	**4.49**	**0.65**

TABLE V
LANE-384/512 COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	# Parallel P_i units	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3						
#15	1	6635	41.0	63	0.66	**0.10**
#16	2	17499	30.4	35	**0.88**	0.05
#17	6	-	-	-	-	-
Virtex-5						
#15	1	3439	142	63	2.30	0.67
#16	2	4030	116	35	3.39	**0.84**
#17	6	14649	69.8	17	**4.20**	0.28

numbers of parallel permutation lanes. For each f_L design, the round constants were pre-computed and stored in Block RAM on the FPGA.

The results are given in Tables IV and V, for LANE-224/256 and LANE-384/512 respectively. Clearly, the more P_i blocks that are implemented in parallel, the better the throughput that is attained. Note that LANE-384/512 would not fit on the Spartan-3 FPGA when 6 P_i permutation units are implemented in parallel. On the Spartan-3, the best throughput per slice is obtained when using just one P_i unit.

VI. SHABAL

Shabal was submitted by the Saphir research project to the SHA-3 contest [14]. It uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits. The hash function output is of size ℓ_h bits, where $\ell_h \in \{192, 224, 256, 384, 512\}$.

A. Specification

The Shabal compression function, denoted here by f_{Sh}, is based on a Non-Linear Feedback Shift Register (NLFSR) construction. This compression function operates on an internal state, denoted by (A, B, C). In the submission to the SHA-3 contest, A is defined as a 12×32-bit word, and B and C are defined as 16×32-bit words. Therefore, Shabal has an

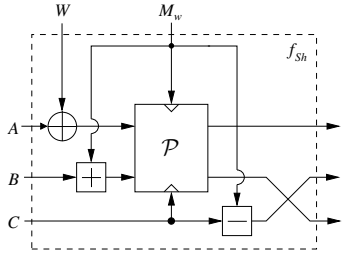

Fig. 8. Shabal Compression Function

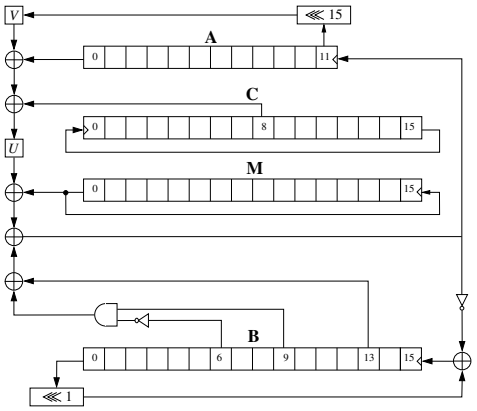

Fig. 9. NLFSR-based Shabal Permutation

internal state of $1,408$ bits. Furthermore, a 64-bit counter W is defined to track the number of message blocks that have been processed. Shabal uses the following operations to compute the hash of a message M:

- Initialisation. The internal state (A, B, C) is initialised to (A_0, B_0, C_0), and this value can be pre-computed.
- Message Padding and Parsing. The message M padded and parsed into N 512-bit blocks M_w, $0 \leq w \leq N-1$.
- Message Compression using f_{Sh}. Each message block M_w is combined with state words B and C using addition and subtraction modulo 2^{32}. State word A is XORed with the counter W. Additionally, a NLFSR-based permutation $\mathcal{P}$ is applied to state words A and B, and $\mathcal{P}$ is parameterised by M_w and C. A block diagram of the compression function f_{Sh} is shown in Fig. 8.
- Finalisation and Output. After block M_{N-1} has been processed, the compression function f_{Sh} is invoked three more times, without changing the value of $M_w = M_{N-1}$ or without updating the counter W. The hash output is formed using ℓ_h bits from state word C.

The permutation $\mathcal{P}$ operates at the core of the Shabal compression function f_{Sh}. To describe $\mathcal{P}$, we use the notation $A[\alpha]$, $0 \leq \alpha \leq 11$, and $B[\beta]$, $C[\beta]$, $0 \leq \beta \leq 16$, to denote a particular 32-bit word of the state. The operation $x \lll y$ denotes bitwise rotation of x by y positions, the operator $\wedge$ denotes Boolean AND, and the Boolean inverse of z is denoted by $\overline{z}$. Operators U and V denote multiplication modulo 2^{32} by integer values 3 and 5, respectively. The permutation $\mathcal{P}$ has three sequential operations:

1) Rotation:
   $B[i] \leftarrow B[i] \lll 17$
   for $0 \leq i \leq 15$;
2) NLFSR-based Permutation:
   a) $A[i + 16j \bmod 12] \leftarrow U(A[i + 16j \bmod 12]$
   $\oplus\, V(A[(i-1) + 16j \bmod 12] \lll 15)$
   $\oplus\, C[8 - i \bmod 16]) \oplus B[i + 13 \bmod 16]$
   $\oplus\, (B[i + 9 \bmod 16] \wedge \overline{B[i + 6 \bmod 16]} \oplus M[i]$
   b) $B[i] \leftarrow (B[i] \lll 1) \oplus \overline{A[i + 16j \bmod 12]}$
   for $0 \leq j \leq 2$ and $0 \leq i \leq 15$; and
3) Addition:
   $A[j \bmod 12] \leftarrow A[j \bmod 12] \boxplus C[j + 3 \bmod 16]$
   for $0 \leq j \leq 35$.

The central part of $\mathcal{P}$ is illustrated in Fig. 9.

TABLE VI
SHABAL COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	Final Additions in $\mathcal{P}$	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3						
#18	Series	1933	89.71	85	0.54	0.27
#19	Parallel	2223	71.48	49	**0.74**	**0.33**
Virtex-5						
#18	Series	2307	222.22	85	1.33	**0.58**
#19	Parallel	2768	138.87	49	**1.45**	0.52

B. Compression Function Implementation & Results

When designing FPGA-based hardware for f_{Sh}, the XOR, addition and subtraction operations were all implemented in parallel. In the permutation $\mathcal{P}$, the rotation operations were implemented through simple wiring. In order to realise the central part of the permutation, we adopted a shift-register based approach, as shown in Fig. 9, where the state words are shifted along chains of 32-bit registers. The multiplication operations U and V form the non-linear part of the NLFSR; these were implemented using the shift-then-add method. Once the shift registers have been loaded with the appropriate initial values, the central permutation result is calculated after 48 clock cycles.

The final part of the permutation $\mathcal{P}$ adds words from the A and C states. For these modulo 2^{32} additions, we investigated two design choices. In the first design, the NLFSR is used together with a single adder to compute $A[0] \boxplus C[3]$, and the result is fed back to $A[15]$. Note that the direction in which the C word is shifted must be reversed. This design takes a further 36 clock cycles to produce the final result. The second design for this stage of $\mathcal{P}$ expands the addition into 12×3 series additions, e.g. $A[0] \leftarrow A[0] \boxplus C[3] \boxplus C[15] \boxplus C[11]$. In this way, the final result is computed without requiring extra clock cycles, but at the expense of area for 35 additional adders.

The area, timing and throughput results for our f_{Sh} implementations are shown in Table VI. In the lower-area implementations, the critical path is within the NLFSR construction, from register $A[11]$ to register $B[15]$. The higher-area implementations have a longer critical path, due to the three series additions used to compute the final result. However, these higher-area implementations still attain better throughputs than the lower-area implementations, due to the lower number of clock cycles required. Both designs have similar throughput per slice metrics, with the higher-area implementation more efficient on the Spartan-3 platform, and the lower-area implementation more efficient on the Virtex-5 platform.

VII. SPECTRAL HASH

Spectral Hash, also called 's-hash', was submitted by Koç et al. to the SHA-3 contest [15], [16]. It is a Merkle-Damgård based function that uses the Discrete Fourier Transform (DFT) to generate the required diffusion and confusion properties.

A. Specification

During the algorithm's execution, the compression function (denoted here by f_{Sp}) operates on the current message block M_i; the hash of the previous message H_{i-1}; and a permutation state P_i, which is dependent on previously processed message blocks. These internal states are termed *prisms* and are represented as $4 \times 4 \times 8$ matrices of different word sizes. Spectral Hash uses the following operations to compute the hash of a message M:

- Message Padding and Initialisation. The message M is padded and parsed into t 512-bit blocks $M_1, \ldots, M_t$. A *p-prism* is created with 7-bit words, and initialised sequentially with the values $0, \ldots, 127$. The words in *p-prism* are then swapped according to the values of words in the first message block M_1. The hash of the previous state is set to the all-zero vector, and stored in *h-prism*.
- Message Compression. Block M_i is placed into an *s-prism* of 4-bit words, and processed with H_{i-1} and P_i.
- Finalisation and Output. When all t message blocks have been processed, a bit marking scheme is utilised to select the required number of bits from the *h-prism*, to form the message digest.

A block diagram of the Spectral Hash compression function f_{Sp} is shown in Fig. 10. It consists of the following operations on both the *s-prism* and the *p-prism*:

- Affine Transformation, performs an inversion in $GF(2^4)$ followed by a linear shift on each byte in *p-prism*.
- Swap Controls, swap bytes in *p-prism* according to the values in *s-prism*.
- Discrete Fourier Transforms, sixteen 8-point DFTs and two sets of thirty-two 4-point DFTs are performed on *s-prism*. Each DFT is performed over the field $GF(17)$.
- Non-linear Transformation, a transformation of *s-prism* according to the state of *p-prism*. The previous hash value *h-prism* is also taken into account in this step.
- Plane Rotation, an output rotation of the *p-prism*.

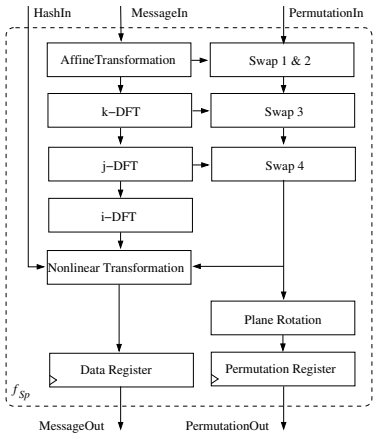

Fig. 10. Block diagram of the Spectral Hash compression function

B. Compression Function Implementation & Results

Two implementations of Spectral Hash were designed. Both implementations use slice Look-Up-Tables (LUT) for the affine transformation as well as the modular 17 reduction in the DFT calculations. The 8-point DFT hardware can also be re-used for the 4-point DFT by padding the extra points with zeros. The *Single Cycle* implementation completes the compression function in a single clock cycle by performing the

TABLE VII
SPECTRAL HASH COMPRESSION FUNCTION IMPLEMENTATION RESULTS

Design Number	Architecture	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Gbps)	TP/Area (Mbps/slice)
Spartan-3						
#20	Low Area	7393	46.50	339	0.07	0.009
#21	Single Cycle	Single-Cycle design too large for Spartan-3 FPGA				
Virtex-5						
#20	Low Area	9051	125.26	339	0.18	0.02
#21	Single Cycle	14601	31.34	1	**16.04**	**1.09**

s-prism and *p-prism* calculations in parallel and fully unrolling the operations. This leads to a large area, i.e. sixteen 8-point DFT hardware architectures are required. The *Low Area* architecture reuses the hardware, saving on area at the expense of extra clock cycles. The results are given in Table VII, which show that on the Virtex-5, very high throughputs can be attained using a fully parallel implementation.

VIII. COMPARISON OF IMPLEMENTATIONS

In order to compare the various hash function implementations with one another, the throughput and area results for all of the designs presented in this paper are plotted in Figs. 11 and 12, for the Spartan-3 and Virtex-5 FPGAs respectively. In each figure, the results are labelled 1–21, corresponding to the labels in Tables I–VII. The crosses ($\times$) denote designs that produce only 224/256-bit hashes; the stars ($*$) denote designs that produce only 384/512-bit hashes; and the bullets ($\bullet$) denote designs that can produce all of the required hash lengths.

The 224/256-bit hash function producing the best throughput per slice on the Spartan-3 FPGA is design #5, i.e. Grøstl-224/256, with S-boxes implemented in BRAM and the P and Q permutations in parallel. The most efficient design on the Virtex-5 FPGA for producing 224/256-bit hashes is design #4, i.e. Grøstl-224/256, with S-boxes implemented in BRAM and the P and Q permutations interleaved. The 384/512-bit hash function producing the best throughput per slice on the Spartan-3 FPGA is design #8, i.e. Grøstl-384/512, with S-boxes implemented in BRAM and the P and Q permutations interleaved. The most efficient design on the Virtex-5 FPGA for producing 384/512-bit hashes is design #21, i.e. single-cycle Spectral Hash (an outlier, not shown on the graph in Fig. 12). In conclusion, of the five hash functions studied in this paper, the Grøstl implementations currently give the best overall balance between throughput and area, when implemented on FPGAs.

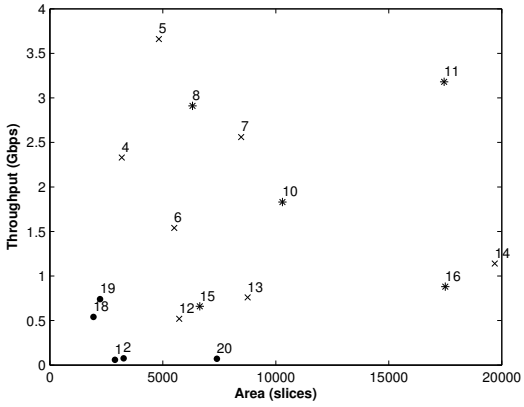

Fig. 11. Throughput vs. Area for the Spartan-3 designs

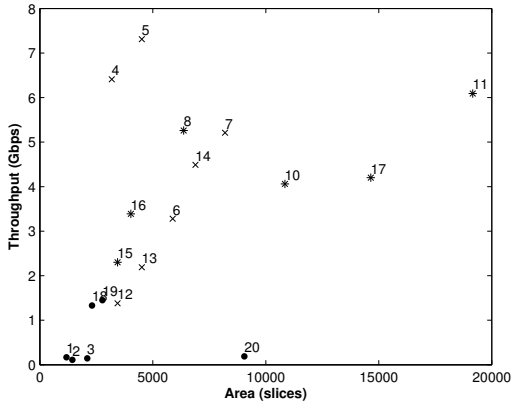

Fig. 12. Throughput vs. Area for the Virtex-5 designs

REFERENCES

[1] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[2] National Institute of Standards and Technology, "FIPS PUB 180-2. Secure Hash Standard," August 2002.

[3] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *Advances in Cryptology — CRYPTO 2005, 25th Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 17–36.

[4] A. K. Lenstra, "Further progress in hashing cryptanalysis (white paper)," http://cm.bell-labs.com/who/akl/hash.pdf, February 2005.

[5] National Institute of Standards and Technology, "Cryptographic hash algorithm competition," http://www.csrc.nist.gov/groups/ST/hash/sha-3/.

[6] ——, "FIPS PUB 197. Advanced Encryption Standard," November 2001.

[7] ——, "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family," *Federal Register*, vol. 72, no. 212, pp. 66 212–66 220, November 2007.

[8] IAIK, "SHA-3 hardware implementations," http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations.

[9] Z. Chen, S. Morozov, and P. Schaumont, "A hardware interface for hashing algorithms," Cryptology ePrint Archive, Report 2008/529, 2008, http://eprint.iacr.org/2008/529.

[10] D. J. Bernstein, "CubeHash specification (2.B.1)," Submission to NIST, 2008.

[11] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," Submission to NIST, 2008.

[12] S. Indesteege, E. Andreeva, C. De Cannière, O. Dunkelman, E. Käsper, S. Nikova, B. Preneel, and E. Tischhauser, "The LANE hash function," Submission to NIST, 2008.

[13] V. Fischer and M. Drutarovský, "Two methods of Rijndael implementation in reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, ser. LNCS, Çetin Kaya Koç, D. Naccache, and C. Paar, Eds., vol. 2162, 2001, pp. 77–92.

[14] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau, "Shabal, a submission to NIST's cryptographic hash algorithm competition," Submission to NIST, 2008.

[15] G. Saldamli, C. Demirkiran, M. Maguire, C. Minden, J. Topper, A. Troesch, C. Walker, and Çetin Kaya Koç, "Spectral hash," Submission to NIST, 2008.

[16] R. C. C. Cheung, Çetin Kaya Koç, and J. D. Villasenor, "An efficient hardware architecture for spectral hash algorithm," Submitted to ASAP 2009.