FPGA Implementations of the Round Two SHA-3 Candidates

Brian Baldwin[†], Neil Hanley[†], Mark Hamilton[†], Liang Lu[††],
Andrew Byrne[‡], Maire O'Neill[††] and William P. Marnane[†]
[†]Claude Shannon Institute for Discrete Mathematics, Coding and Cryptography
Department of Electrical & Electronic Engineering
University College Cork, Ireland
{brianb,markh,neilh,liam}@eleceng.ucc.ie
[‡]School of Mathematical & Geospatial Sciences,
RMIT University, Melbourne, Australia
andrew.byrne@rmit.edu.au
[††]The Institute of Electronics, Communications & Information Technology
Queen's University Belfast, Belfast, UK
{l.lu, m.oneill}@ecit.qub.ac.uk

August 27, 2010

Abstract

The second round of the NIST-run public competition is underway to find a new hash algorithm(s) for inclusion in the NIST Secure Hash Standard (SHA-3). This paper presents full hardware implementations of all of the second round candidates in hardware for all specified message digest variants. In order to determine their computational efficiency, a specified aspect in NIST's round two evaluation criteria, this paper gives an area/speed comparison of each design both with and without a hardware interface, thereby giving an overall impression of their performance in resource constrained and resource abundant environments. The post-place-and-route implementation results are provided for a Virtex-5 FPGA device. The efficiency of the architectures for the hash functions are compared in terms of throughput per unit area.

To the best of the authors' knowledge, this is the first work to date to present hardware designs which test for all message digest sizes (224, 256, 384, 512), and also the only work to include the padding as part of the hardware for the SHA-3 hash functions.

1 Introduction

The NIST hash competition [1] to select a new hash algorithm(s) for the purpose of ultimately superceding the functions in the SHA-2 family is currently nearing the end of the second round evaluation period. The fourteen contesting designs for SHA-3 (or the Advanced Hash Standard (AHS)) which advanced to round two are available for public comment and scrutiny, and 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 [1]. The work in this paper is a continuation of work in [2], in which three of the five selected hash functions progressed to round two, CubeHash, Grøstl

and Shabal. We present updated round two results for these implementations along with baseline implementations for each of the other second round designs.

The rest of this paper is organised as follows. In Section 2 we give a brief description of our design decisions and our testing methodology. Section 3 gives an overview of hash functions and the hash function architectures, with subsections 3.4–3.13 respectively describing the FPGA implementations of the 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 Virtex-5 FPGA platform are supplied. Section 4 gives results for the designs, and Section 5 concludes.

2 Fair Comparison Methodology

In the NIST competition specifications [1], $6.C$, *Round 2 Technical Evaluation* gives the criteria for hardware testing; *"Round 2 testing by NIST will be performed on the required message digest sizes"* and *"the calculation of the time required to compute message digests for various length messages"*.

As such, the authors felt that for a complete analysis of the hash functions as required by NIST, it was necessary to implement as many designs as was necessary for full coverage of all of the message digest sizes, {224, 256, 384, 512}. While in some cases, all four variants can be output from a single design, where only the initial vectors (IV) and truncation differ, others require two or even three different designs to produce the four output digests. The current hash standard, SHA-2, was also implemented as a reference point. To allow fair comparison between different designs, all designs were implemented in slice logic, using distributed memory instead of dedicated block memory where required. Although this does not make the best use of the FPGA, all the designs and their variants can thus be fairly measured and analysed.

For the calculation time, it was decided to also include the padding stages in hardware, required both for the purpose of testing the hash functions against the Known Answer Tests (KAT) values provided by each designs submission package, and also to give a fair and accurate timing report inclusive of all stages required to hash a message. As there are a number of different padding schemes, each of which have differing number of rounds required to complete, the padding also has a bearing on the calculation time.

We developed a hardware wrapper interface [3] which necessarily needed to be published separately due to space limitations. This wrapper, freely available on our website, along with all other source code, allows others to test their designs, `http://www.ucc.ie/en/crypto/`, shown in Fig. 1, produces the padding scheme required for a particular hash function as well as providing the interface to the outside world. It allows re-use of any padding scheme that can be used in multiple hash functions.

In this case we set the i/o bus widths, w, to 32-bits, a standard word size. This models a realistic communications system, and takes into account any bandwidth limitations, as any hash function requiring a large message size, m, will be subject to a latency of m/w clock cycles. Our hash functions necessarily take this latency into account, and we implemented our designs so as to minimise where possible any delays due to this loading of data, i.e. the hash round function should, if possible, take longer than the time required to load a message. We showed in [3] that, while the wrapper itself does not affect the clock frequency of the design, counters in the padding block may form the critical path and thus affect the timing.

As such, in the interest of fairness while our main results present area and frequency results inclusive of the wrapper, we also present an appendix section detailing the hash results where the padding is implemented in software, both exclusive of the wrapper, thereby negating any transmission bottlenecks, and inclusive of the wrapper to allow a fair comparison.

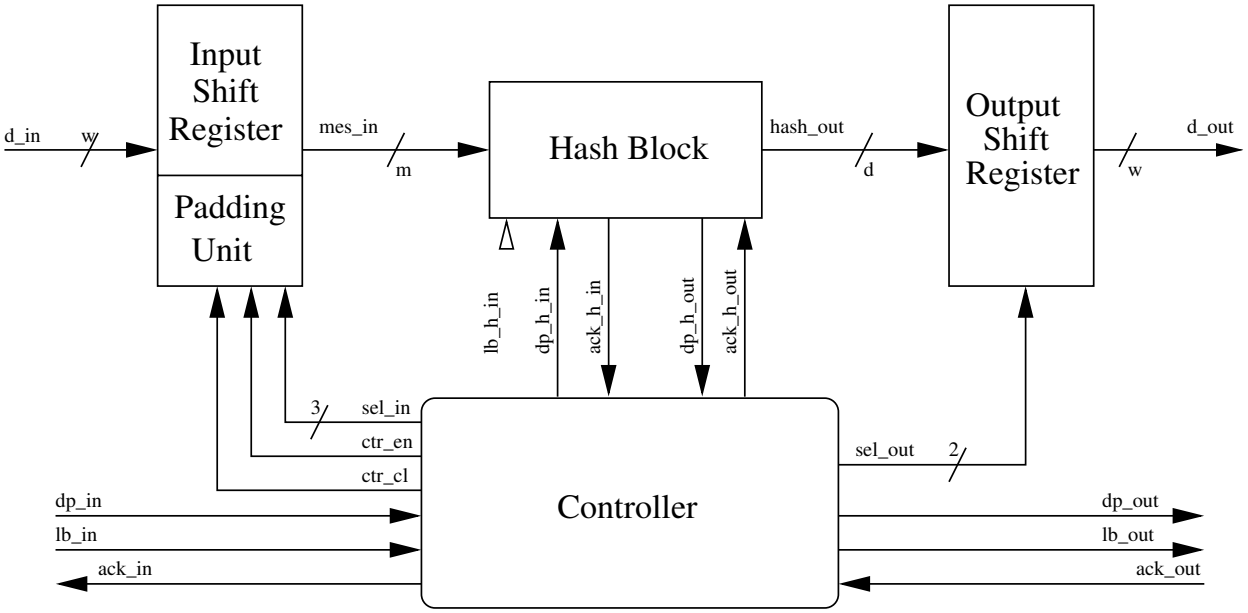

Figure 1: Wrapper Interface

3 Overview of the Hash Function Architectures

Table 1 gives the constructions of the different SHA-3 hash functions and their variants as well as the various inputs and state sizes in bits. As the {224} variant is almost identical to the {256} and similarly, the {384} to the {512} we omit these values. The only notable differences being Keccak, where the message size increases to 1152-bits for {224}

Table 1: Hash Function Internals

			224/256				384/512			
Design	Structure	Type	Counter	Message	Salt	State	Counter	Message	Salt	State
SHA-2	Merkle-Damgård	Add-XOR-Rotate	64	512	-	512	128	1024	-	1024
Blake	HAIFA	Add-XOR-Rotate	64	512	128	512	128	1024	256	1024
BMW	Iterative	Add-XOR-Rotate	64	512	-	2048	64	1024	-	4096
Cubehash	Iterative	Add-XOR-Rotate	-	256	-	1024	-	256	-	1024
Echo	HAIFA	AES based	64	1536	128	2048	64	1536	128	2048
Fugue	Iterative	AES based	64	32	-	96-	64	32	-	1148
Grøstl	Iterative	AES based	64	512	-	512	64	1024	-	1024
Hamsi	Conc-Permute	Serpent based	64	32	-	512	64	64	-	1024
JH	Iterative	Block Cipher based	128	512	-	1024	128	512	-	1024
Keccak	Sponge	Add-XOR-Rotate	-	1088	-	1600	-	576	-	1600
Luffa	Sponge	S-box based	-	256	-	768	-	256	-	1280
Shabal	Iterative	Add-XOR-Rotate	-	512	-	1408	-	512	-	1408
SHAvite-3	HAIFA	AES based	64	512	256	256	128	1024	512	512
SIMD	Iterative	Block Cipher based	64	512	-	512	64	1024	-	1024
Skein	UBI	Add-XOR-Rotate	96	512	-	512	96	512	-	512

and 832-bits for {384}, and Luffa, where the state size decreases to 1024-bits for {384}. The *Structure* loosely defines the hash function overview, for example, in HAIFA (Hash Iterative Framework) [4] based designs, the counter is fed in with the message, whereas, for Merkle-Damgård [5] [6], it is not. The *Type* describes the design of the hash functions. The *Counter*, *Message* and *Salt* all form the inputs to the hash functions, while the *State* describes the internal size of each of the hash functions.

Table 2 gives the different padding schemes used by the hash functions. There are many different padding schemes utilised by the designers of the hash functions, and in some cases varying padding schemes between the different sizes of the same hash function. As can be seen from the Table, similarities between some of the different padding schemes allow us to generate a generic block for variants of Merkle-Damgård strengthening [7] padding schemes, as well as paddings types of all-zeros or one-and-trailing-zeros.

Understandably this review is somewhat brief and we invite the reader to review the SHA-3 submission documentation for a full description of each of the hash functions.

Table 2: Padding Schemes

Design	Padding Scheme
SHA224/256	1, 0's until congruent (448 mod 512), 64-bit message length
SHA384/512	1, 0's until congruent (896 mod 1024), 128-bit message length
Blake224	1, 0's, until congruent (448 mod 512), 64-bit message length
Blake256	1, 0's, until congruent (447 mod 512), 1, 64-bit message length
Blake384	1, 0's, until congruent (895 mod 1024), 128-bit message length
Blake512	1, 0's, until congruent (894 mod 1024), 1, 128-bit message length
BMW224/256	1, 0's until congruent (448 mod 512), 64-bit message length
BMW384/512	1, 0's until congruent (960 mod 1024), 64-bit message length
Cubehash	1, 0's until a multiple of 256 (256 = 8 * b, b=32)
Echo224/256	1, 0's until congruent (1392 mod 1536), 16-bit message digest, 128-bit message length
Echo384/512	1, 0's until congruent (880 mod 1024), 16-bit message digest, 128-bit message length
Fugue	0's until a multiple of 32, 64-bit message length
Grøstl224/256	1, 0's until congruent (448 mod 512), 64-bit block counter
Grøstl384/512	1, 0's until congruent (960 mod 1024), 64-bit block counter
Hamsi224/256	1, 0's until a multiple of 32, 64-bit message length
Hamsi384/512	1, 0's until a multiple of 64, 64-bit message length
JH	1, 0's until congruent (384 mod 512), 128-bit message length, min 512-bits added
Keccak224	1, 0's until a multiple of 8, append 8-bit representation of 28, append 8-bit representation of 1152/8, 1, 0's until a multiple of 1152
Keccak256	1, 0's until a multiple of 8, append 8-bit representation of 32, append 8-bit representation of 1088/8, 1, 0's until a multiple of 1088
Keccak384	1, 0's until a multiple of 8, append 8-bit representation of 48, append 8-bit representation of 832/8, 1, 0's until a multiple of 832
Keccak512	1, 0's until a multiple of 8, append 8-bit representation of 64, append 8-bit representation of 576/8, 1, 0's until a multiple of 576
Luffa	1, 0's until a multiple of 256
Shabal	1, 0's until a multiple of 512
SHAvite3-224/256	1, 0's until congruent (432 mod 512), 64-bit message length, 16-bit digest length
SHAvite3-384/512	1, 0's until congruent (880 mod 1024), 128-bit message length, 16-bit digest length
Simd224/256	0's until a multiple of 512, extra block with message length
Simd384/512	0's until a multiple of 1024, extra block with message length
Skein	0's if multiple of 8, else 1, 0s, until a multiple of 512

In the design of the hash function architectures described in this paper, our main goal was to give a baseline comparison between the hash functions using area and throughput. We calculate the throughput as follows:

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

The FPGA platform targeted in the study was the Xilinx Virtex-5 *xc5vlx330T-2-ff1738*. Each hash function design was implemented using VHDL, and Synthesis, Place and Route were carried out using Xilinx ISE v9.2i. We measure area of our hash function designs in FPGA slices, as given by the Map report.

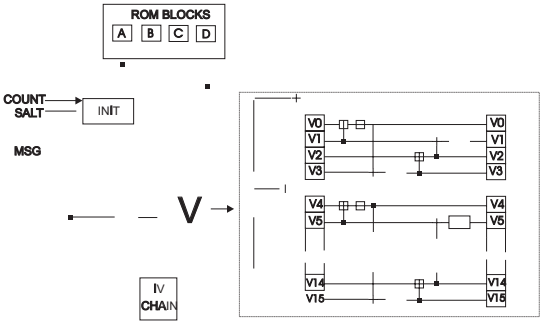

Figure 2: Blake

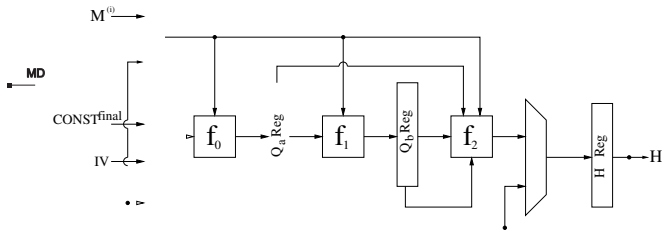

Figure 3: BMW

3.1 BLAKE - Aumasson *et al.*

For our implementation of BLAKE we further subdivide the compression function into two identical sections, to allow re-use of the component blocks and thereby reducing the area. This subdivision increases the latency of the hash permutation to complete a round from two to four clock cycles, but reduces the critical path from four adders to two adders thus increasing the maximum frequency of the permutation. For the larger variant which requires 32 clocks to load a 1024 bit message, it ensures there is no delay where the hash function needs to wait for loading to complete. Fig. 2 shows the modified design.

The Adders and XORs are generated using standard operators (using the '+' operator of the IEEE.std logic unsigned package) and the rotation operations were implemented through simple wiring, with multiplexers to select the particular subround rotation. The 16 constants required by the initialisation and round stages are stored in distributed ROM.

3.2 Blue Midnight Wish - Gligoroski *et al.*

Our implementation of Blue Midnight Wish (BMW) is designed as follows. f_0 takes $M^{(i)}$ and $H^{(i-1)}$ as its inputs and produces the first half of the quadrupled-pipe value $Q_a^{(i)}$. f_0 consists of 80 additions/subtractions, as well as XOR's, bitwise shifts and rotations. f_1 takes $M^{(i)}$, $H^{(i-1)}$ and $Q_a^{(i)}$ as its inputs and produces the second half of the quadrupled-pipe value $Q_b^{(i)}$. The quadrupled-pipe is then $Q^{(i)} = \left(Q_a^{(i)}, Q_b^{(i)}\right)$. f_1 is the most complex of the functions performed by BMW, consisting of two sub functions ER_1 and ER_2. Both ER_1 and ER_2 contain sixteen modulo 32 addition operations but ER_1 contains more bitwise shift and rotate operations. Both functions use an operation that uses modulo 32 additions, subtractions and rotations to combine a block of the message and of the double-pipe with a predefined set of constants. The final function f_2 takes $M^{(i)}$, $Q_a^{(i)}$ and $Q_b^{(i)}$ as inputs and produces the new double-pipe value $H^{(i)}$. f_2 consists of XOR, bitwise shift, rotation and modulo 32 addition operations.

A pipelined design was chosen for implementation due to the large amount of additions that need to be performed. Each of the functions f_0, f_1 and f_2 make up a stage in the pipelined deign. One operation of the compression cycle therefore takes three clock cycles. A diagram of the pipelined designed is shown in Figure 3.

3.3 CubeHash - Bernstein

We designed FPGA implementations of the CubeHash compression function with round two parameters as recommended by Bernstein in the round 2 tweaks. The rotation and swapping operations are implemented in hardware by

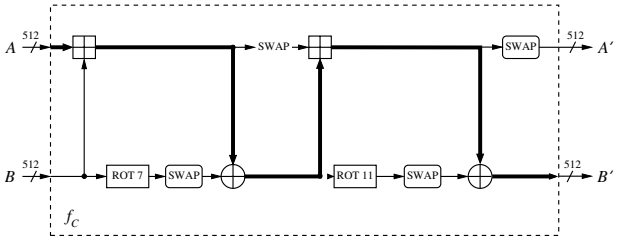

Figure 4: Cubehash

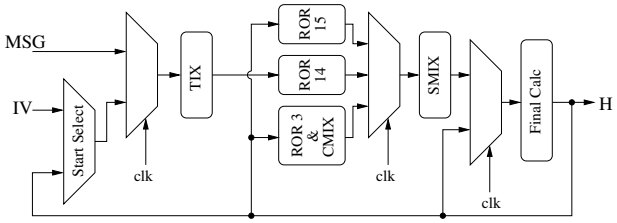

Figure 5: Fugue 256

simply re-labelling the relevant signals. Since the state comprises 1024 bits, the same architecture can be used to produce message digests with any of the lengths required for SHA-3. Therefore, a CubeHash8/32-256 implementation will have the same throughput and throughput per slice performance as a CubeHash8/32-512 implementation.

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. 4. 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 implemented the CubeHash architectures where f_C is unrolled to a chain of four f_C units in series to process a single message block in two clock cycles. Note that the figures quoted for 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.

3.4 ECHO - Orange Labs

The compression function (CF^E) for ECHO, operates iteratively as follows: $V_i = CF^E(V_{i-1}, M_i, C_i, SALT)$ where V_{i-1} is the current value of the chaining variable, M_i is the current message block, C_i is a counter and $SALT$ is a sub-key.

We designed and evaluated FPGA implementations of the ECHO hash function with an output of {256 & 512} bits. In the proposed design, an iterative architecture is used with one *BIG.ROUND* function and a finite state machine (FSM) to control the data-path. Within the *BIG.ROUND* function, 16 pairs of the AES functions are processed in parallel to improve the throughput rate. The S-boxes are implemented using distributed ROM memory. Sub-keys are pre-calculated prior to the compression function. An outline of this architecture is provided in Fig. 6. In this design, the *BIG.ROUND* operation is performed in one clock cycle.

3.5 Fugue - Halevi *et al.*

The Fugue hash function was designed by researchers at IBM. The rotation blocks (ROR) operate on different sizes for each of the variants, but the overall design remains the same. Fig. 5 shows the operation of **F**-256. The initial round comprises a TIX stage (XOR, truncate, insert and XOR of bytes), a rotation by 3 bytes (ROR3) and a column mix (CMIX), all on each of the n blocks, followed by a super mix (SMIX) transformation. This transformation takes a 4x4 matrix of bytes and passes each byte through an S-box, followed by a linear transformation to generate diffusion. This linear transformation is similar to that employed in AES, however, unlike AES, there is cross-mixing between the columns. These steps are looped a number of times r depending on the variant, with $r = 2, 3, 4 \in \{224, 256, 384, 512\}$ per message block. The final round comprises two more loops of a rotation by three bytes (ROR3), CMIX and SMIX (repeated p times), an XOR, and a rotation by 15 or 14 followed by an SMIX (repeated q times). For the different variants $p = 5, 18, 32 \in \{224, 256, 384, 512\}$ and $q = 13 \in \{224, 256, 384, 512\}$.

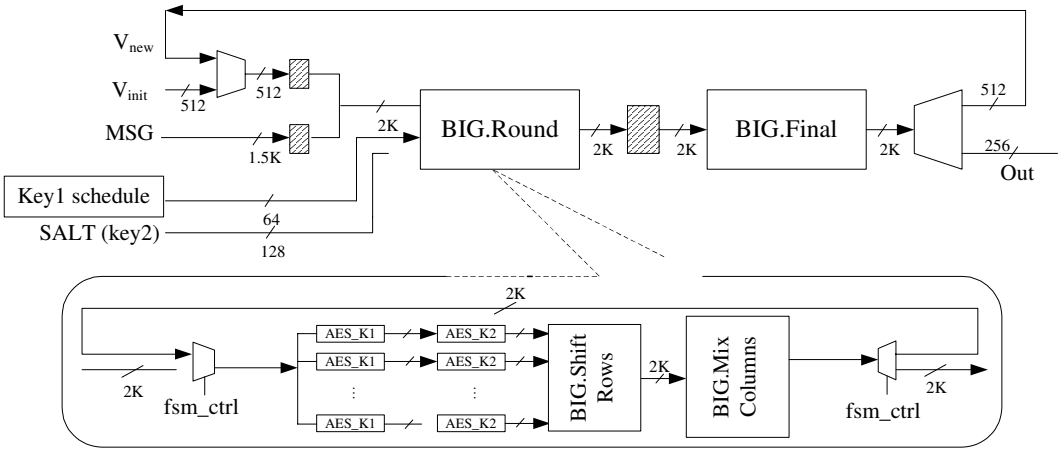

Figure 6: Echo

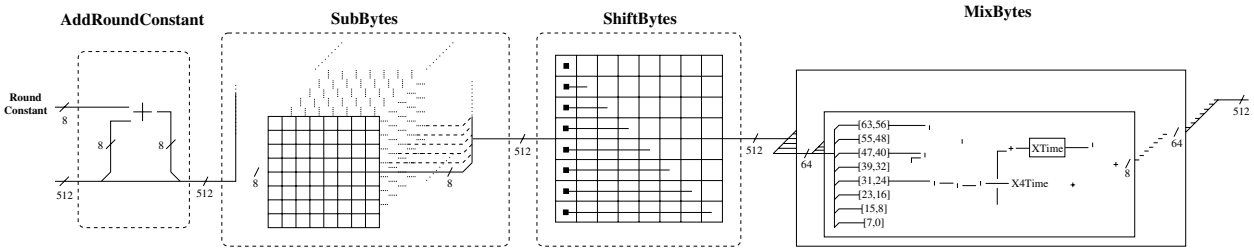

Figure 7: Groestl

In all variants, the CMIX and linear transformation operations are implemented using combination logic, and the rotation operation was implemented through rewiring. As defined in the specifications, for each design we declared four blocks where the state words are operated on and shifted along chains of 8-bits. The S-boxes are implemented as look up tables using distributed ROM memory.

3.6 Grøstl - Gauravaram *et al.*

The architecture for Grøstl is illustrated in Fig. 7. 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 generated using distributed ROM. 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. An output register was used to store the state at the output of the MixBytes transformation.

The compression function f_G for the Grøstl implementation consists of two permutation functions, P and Q. Permutations P and Q are identical except for the execution of the AddRoundConstant step, where different round constants are used. Therefore, our design choice was to compute Q in parallel by replicating the hardware for P. Two

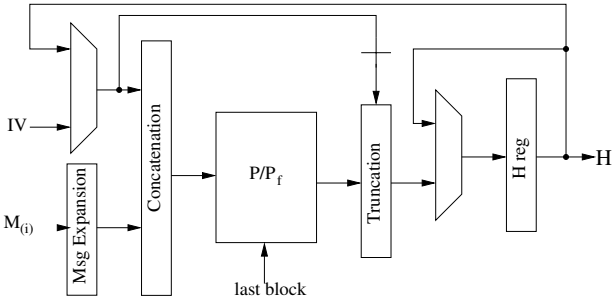

Figure 8: Hamsi

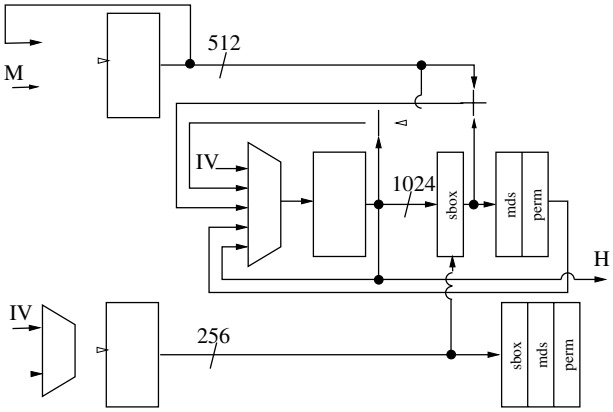

Figure 9: JH

XOR arrays are required to complete the compression function for the input to P, and for the final output H_i.

3.7 Hamsi - Küçük

The Hamsi hash function has a concatenate - permute - truncate construction, with the input message expanded and concatenated with an initial value or the output from the previous stage of the hash function. This is followed by the non linear permutation, made up of XORing the state with a table of predefined constants and a counter, Serpent [8] S-boxes and a diffusion operation consisting of several bitwise shifts and XOR's. Truncation reduces the Hamsi state down to the size of the input message.

A fully parallel design was chosen for implementation as shown in Figure 8. The non-linear permutation *P* was unrolled three times, more unrolling resulted in a congested design for the VHDL. Therefore it takes one clock cycle for a normal message block to be hashed and two clock cycles for the final message block. The Serpent based S-boxes were generated using distributed ROM.

3.8 JH - Hongjun Wu

JH uses the same design for all four varients and is based on simple components. The compression function combines a 1024-bit previous hash block (H_{i-1}), a 512-bit message block (M_i) to produce a 1024-bit hash block (H_i). The compression function (CF^{JH}) is applied to each message block, M_i. The bijective function consists of 35 rounds, each consisting of an S-box, linear transformation and permutation, and a single final round consisting of just the S-box. Two 4-bit S-boxes are used, the selected table depending on the value of a round constant. It can also be viewed as a 5-bit to 4-bit substitution. The linear transformation implements a $(4, 2, 3)$ maximum distance separable (MDS) code over $GF(2^4)$, and the permutation shuffles the output according to three distinct smaller permuations. The 256-bit round constants can be generated either in parallel with the data path or pre-computed and stored in memory where they can be re-used. In the design presented, the full 1024-bit data state is operated on at once. Each round completes in one clock cycle. The 256-bit sub-key state is calculated in parallel, as illustrated in Fig. 9.

The S-box and linear transformation functions are implemented as combinational logic as outlined in the submission documentation, and the grouping and permutation functions are rewiring circuits. In the round constant data path,

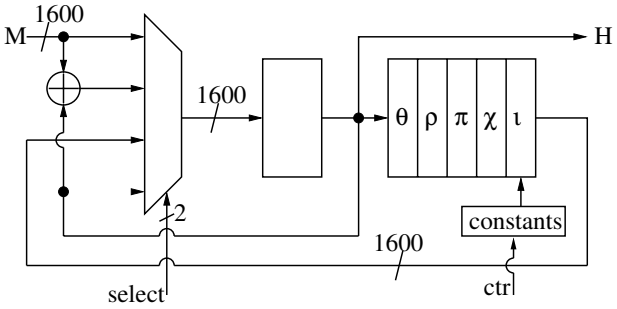

Figure 10: Keccak *f(1600)*

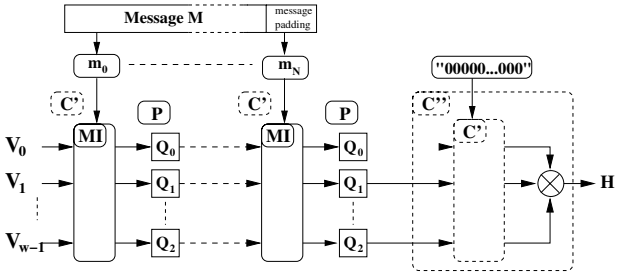

Figure 11: Luffa

only the S-box corresponding to select bit '0' is required. Three registers are required for data storage, one each for the round constant, message block and data block respectively.

3.9 Keccak - Bertoni *et al.*

Keccak is a hash function based on the sponge construction [9]. The NIST submissions use the same KECCAK-f permuation for all variants, with different capacity (c), bitrate (r) and diversifier (d) values, where smaller digest sizes have a greater bitrate. The five steps of the permutation consist of addition and multiplication operations in $GF(2)$. The full round computes in a single clock cycle, and an extra clock is required for loading in of the message. The padded message of length r is loaded in by XOR'ing it with r bits of the state. The 64-bit round constants are defined as the output of a linear feedback shift register and can be pre-computed or generated as required. In the design presented, they are pre-computed and stored in distributed ROM. Only one register is required and is used to store the state value. The HDL implementation provided in the specification documentation was used as a reference for the permutation steps in our design.

3.10 Luffa - De Cannière *et al.*

During the round function of Luffa, a message injection and permutation function are applied to these inputs as illustrated in Fig. 11. The round function consists of a message injection function (MI) and a permutation function (P). The MI can be implemented simply using an array of XOR gates as defined in the specifications. The round function is more complex and consists of w non-linear permutation functions, Q_j, which execute 8 iterations of a step function, where $w = 3, 4, 5 \in \{224/256, 384, 512\}$. Each Q_j performs an input tweak function followed by 8 iterations of the step function. The step consists of an S-box transformation, implemented in distributed ROM. The MixWord function is a linear permutation of two words and is implemented by a series of shifts and XORs. The final stage of the step function is AddConstant in which a predefined step constant is XORed to a single word of the input.

The step constant is dependant on the current iteration of the round function. The core unit in the implementation of Luffa is the Step function which can be executed in a single clock cycle while still maintaining a minimum clock delay. Due to the S-box, a clock delay is incurred here. To reduce any further clock delays, the output of one iteration of the step function is passed directly to the input of the next. In this way, one round of Q_j will take 8 clock cycles. Each Q_j can be executed in series or in parallel in order to target area or speed optimizations. The only differences between each instance are the step constants. Therefore, in order to implement each Q_j using a single instance of Q, a mux is required to control the selection of the constants.

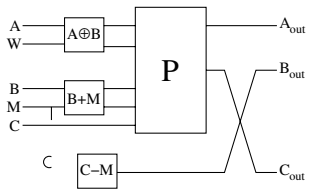

Figure 12: Shabal

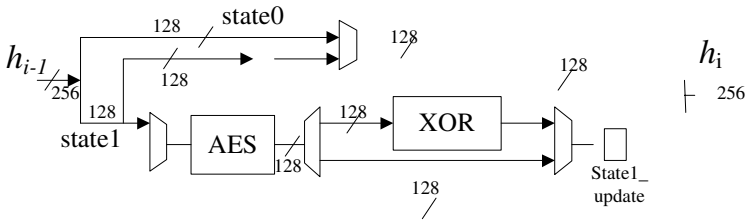

Figure 13: SHAvite-3

3.11 Shabal - Saphir research project

Shabal uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits , as shown in Fig. 12. The Shabal compression function is based on a Non-Linear Feedback Shift Register (NLFSR) construction. We use the precomputed IV to remove the configuration stage and thus remove the initial two message block from the latency. When designing Shabal, 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, 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 expand the addition into 12×3 series additions. Using this approach, the final result is computed without requiring any extra clock cycles, at the cost of 35 additional adders.

3.12 SHAvite-3 - Biham *et al.*

The compression function for SHAvite-3 is a keyed permutation that is used with the Davies-Meyer construction [10]. To achieve a high throughput rate, a fast AES module is needed. Since the AES modules are processed sequentially, only one AES block is required in the compression function module. The architecture is shown in Fig. 13. For a parallel implementation, a second AES block is required for computation. In the compression function architecture, the 256-bit chaining variable is split into two 128-bit parts, namely *state0* and *state1*. the AES function using a 128-bit data bus processes *state1* in one clock cycle. The output of the AES function will be fed back to be processed again. After three AES computations, the output is XORed with the data from the *state0* registers. after the XOR operation *state0* and *state1* update and input into the next round computation. In total, 12 rounds are required in CF_{256}^{S3}. The updated chaining variable or hash output is available after 37 clock cycles.

The AES S-boxes are implemented using distributed ROM memory. In key-expansion, in the non-linear stage 1 clock cycle is required to generate every four keys. In the linear stage 1 clock cycle to generate every eight keys. The total number of clock cycles required for generating all keys is 25. In order to produce a corresponding key ahead of the compression function, the key-expansion module needs to be triggered one clock cycle earlier than the compression function.

3.13 SIMD - Leurent *et al.*

SIMD is an iterated hash function, based on the Merkle-Damgård design, with a modified Davies-Meyer function compression function using a Fiestel-like block cipher. The design of the the compression function using parallel

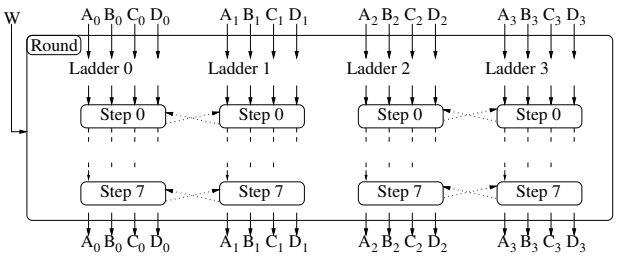

Figure 14: SIMD

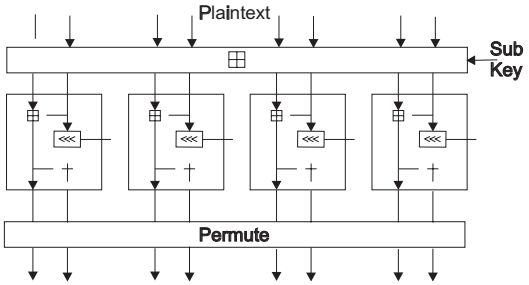

Figure 15: Skein 512

Fiestel ladders allows for high throughput implementations on hardware. The inner state, S, is represented as a 4x4 matrix of 32-bit words for SIMD-256, or an 8x4 matrix for SIMD-512. The function E consists of four rounds of parallel Fiestel Ladders, each composed of eight Fiestel Steps. The Fiestel Step is the core unit of the compression function. There are eight Fiestel Steps in each Ladder, with four Ladders in parallel in each Round (eight Ladders for SIMD-512) as shown in Fig. 14.

Registers are placed on the outputs of each Fiestel Step, as well as on each round, in order to minimize the critical path and therefore maximise the clock frequency. Removing some of these registers, reduces the number of clock cycles required to complete an iteration of the compression function but the effect on the throughput is negated by the reduction of the overall clock frequency. There are a total of four rounds in each compression function, followed by a final *half*-round consisting of four parallel Fiestel Ladders consisting of four Fiestel Steps each.

3.14 Skein 512 - Ferguson *et al*

Skein-512 is the primary proposal of the Skein family of algorithms. A Unique Block Iteration (UBI) chaining mode takes in the chain value, the message and a 'Tweak' defined by an 128-bit configuration string derived from the message counter and UBI constants. The Threefish algorithm has 72 rounds consisting of four sets of four MIX functions followed by a permutation of the eight 64-bit words. Each MIX function consists of a single addition, a rotation by a constant, and an XOR. The rotation constants repeat every eight rounds. The key schedule generates the subkeys from the chain and a tweak. A finalisation UBI stage consisting of a null message, a Tweak and the previous chain.

For our design of Skein-512 we unrolled four rounds of threefish, Fig. 15. In this way, a UBI message block of Skein takes 18 clocks for the rounds to complete, plus 5 for preprocessing and data loading. We use the precomputed IV to remove the configuration stage and thus remove the initial message block from the latency. Each subsequent message block and the output block are calculated identically. The tweak, which ensures each message block is different, is generated by the counter in the padding.The Adders and XORs were generated in a generic fashion and the rotation operations were implemented through simple wiring, with multiplexers to select the particular subround rotation.

4 Results

Table 3 gives the clock count for the various designs. As can be seen from the table, some hash designs require extra time to load in the padding scheme, while others have finalisation stages comprising a number of rounds. For calculating the throughput, as the size of the message to be hashed increases, these padding and finalisation stages

Table 3: Hash Function Timing Results

Hash Design	32-bit load #Cycles	Extra Padding	Padding #Cycles	Message Rounds	Round #Cycles	Long Msg #Cycles	Final Rounds	Final #Cycles	Short Msg #Cycles
SHA224/256	16	0	0	64	1	65	0	0	65
SHA384/512	32	0	0	80	1	81	0	0	81
Blake224/256	16	0	0	10	4	40	0	0	40
Blake384/512	32	0	0	14	4	56	0	0	56
BMW224/256*	16	0	0	1	4	4	1	3	7
BMW384/512*	32	0	0	1	4	4	1	3	7
Cubehash	8	0	0	16	17	17	160	161	178
Echo224/256*	48	0	0	8	1	8	1	1	9
Echo384/512*	32	0	0	10	1	10	1	1	11
Fugue224/256	1	2	1	1	7	7	13	91	98
Fugue384	1	2	1	1	10	10	20	180	190
Fugue512	1	2	1	1	13	13	22	264	277
Grøstl224/256*	16	0	0	10	1	10	0	0	10
Grøstl384/512*	32	0	0	14	1	14	0	0	14
Hamsi224/256	1	3	1	3	2	6	6	24	31
Hamsi384/512	2	3	1	6	2	12	12	48	61
JH	16	1	1	35	1	38	0	0	38
Keccak224*	36	0	0	24	1	25	0	0	25
Keccak256*	34	0	0	24	1	25	0	0	25
Keccak384*	26	0	0	24	1	25	0	0	25
Keccak512	18	0	0	24	1	25	0	0	25
Luffa224/256	8	0	0	8	1	8	1	8	16
Luffa384	8	0	0	8	1	8	2	16	24
Luffa512	8	0	0	8	1	8	2	16	24
Shabal	16	0	0	1	50	50	3	150	200
SHAvite3-224/256	16	0	0	12	3	36	1	1	37
SHAvite3-384/512	32	0	0	14	4	56 (70)	1	1	71
Simd224/256	16	1	1	4	8	32(41)	0.5	4	36(45)
Simd384/512	32	1	1	4	8	32(41)	0.5	4	36(45)
Skein	16	0	0	18	22	22	18	22	44

will have less of an impact on the overall calculation time. However for short messages, they have a big impact. **We therefore define a short message as the time required to process the padding, a single message block and finalisation, and a long message as just the time to process the message block**. Note that each hash function operates over the state size given in Table 1, and so designs with smaller state sizes will require a larger number of rounds to hash the same amount of data as a design with a large state size. This is also reflected in the throughput.

The larger state sizes however are affected by the loading latency as explained in Section 2. Where the time required to hash the message is larger than the time required to load the message this only affects the initial message loading, but in cases where the load latency is longer than the hash latency (denoted * in Tables 3 and 4), there will be a delay as the hash waits for data to load. In this scenario the clock count for the throughput needs to take this additional delay into consideration. Not given here is the output message load time, which in all cases is the hash digest size/output bus size.

The timing and area results for the implemented hash functions on the Xilinx Virtex-5 *xc5vlx330T-2-ff1738* are given in Table 4. The *-w* designation defines the results inclusive of the wrapper, while $-nw$ gives the hash function as a stand alone entity. The throughput results given are inclusive of the wrapper with *TP-s* using Table 3's clock count for a **short message**, and *TP-l* using the clock count for a **long message** where the padding and finalisation stages will have little impact on the message. We highlight SHA-2 as the benchmark to compare the others against. The bar graphs present throughput and throughput/area results for both long and short messages inclusive of the wrapper. An appendix section presents the hash results for the padding implemented in software both inclusive and exclusive of the wrapper.

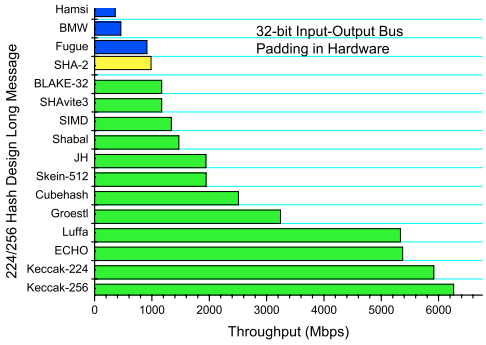

Figure 16: 224/256 Long Throughput

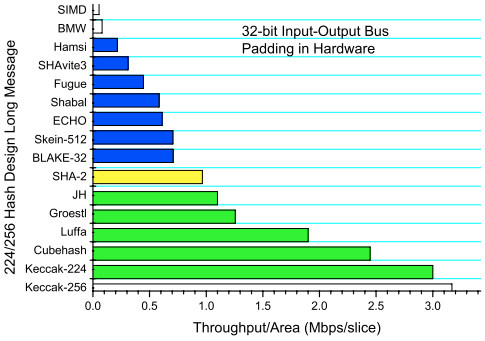

Figure 17: 224/256 Long Tp/Area

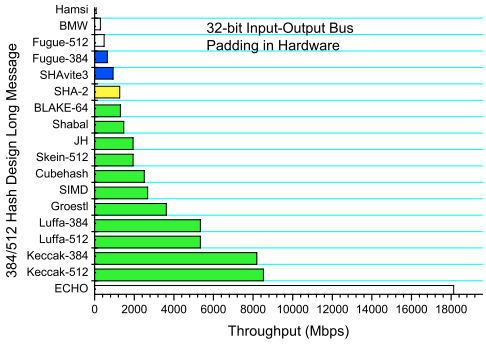

Figure 18: 384/512 Long Throughput

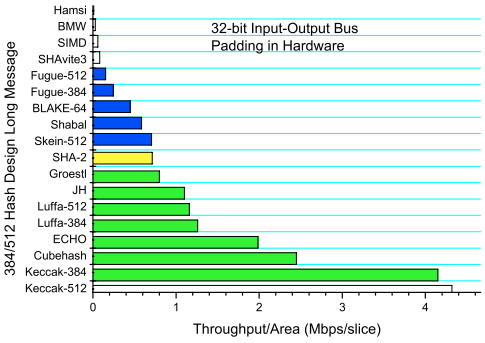

Figure 19: 384/512 Long Tp/Area

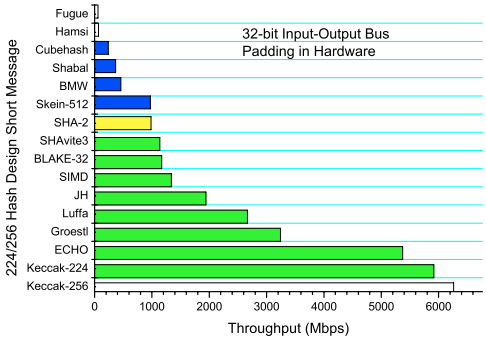

Figure 20: 224/256 Short Throughput

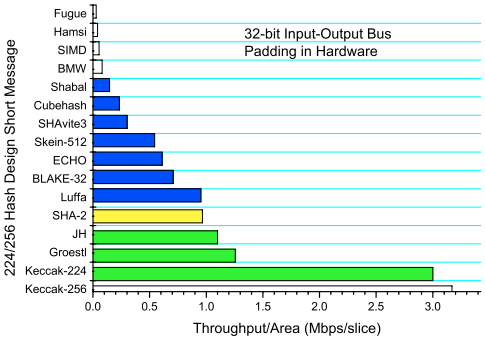

Figure 21: 224/256 Short Tp/Area

Table 4: Hash Function Implementation Results

Hash Design	Area-w (slices)	Max.Freq-w (MHz)	Area-nw (slices)	Max.Freq-nw (MHz)	TP-l (Mbps)	TP-l/Area (Mbps/slice)	TP-s (Mbps)	TP-s/Area (Mbps/slice)
SHA-2-256	1,019	125.063	656	125.125	985	0.966	985	0.966
SHA-2-512	1,771	100.04	1,213	110.096	1264	0.713	1264	0.713
BLAKE-32	1,653	91.349	1,118	118.064	1169	0.707	1169	0.707
BLAKE-64	2,888	71.048	1,718	90.909	1299	0.449	1299	0.449
BMW-256*	5,584	14.306	4,997	14.016	457	0.081	457	0.081
BMW-512*	9,902	8.985	9,810	10.004	287	0.028	287	0.028
Cubehash	1,025	166.667	695	166.833	2509	2.447	239	0.233
ECHO-256*	8,798	161.212	7,372	198.926	5373	0.61	5373	0.61
ECHO-512*	9,130	166.667	8,633	166.694	18133	1.986	5666	0.608
Fugue-256	2,046	200	1,689	200.04	914	0.446	60	0.029
Fugue-384	2,622	200.08	2,380	200.08	640	0.244	33	0.012
Fugue-512	3,137	195.81	2,596	200.16	481	0.153	22	0.007
Grøstl-256*	2,579	78.064	2,391	101.317	3242	1.257	3242	1.257
Grøstl-512*	4,525	113.122	4,845	123.396	3619	0.799	3619	0.799
Hamsi-256	1,664	67.195	1,518	72.411	358	0.215	69	0.041
Hamsi-512	7,364	14.931	6,229	16.51	79	0.01	15	0.002
JH	1,763	144.113	1,291	250.125	1941	1.1	1941	1.1
Keccak-224*	1,971	195.733	1,117	189	5915	3	5915	3
Keccak-256*	1,971	195.733	1,117	189	6263	3.17	6263	3.17
Keccak-384*	1,971	195.733	1,117	189	8190	4.15	8190	4.15
Keccak-512	1,971	195.733	1,117	189	8518	4.32	8518	4.32
Luffa-256	2,796	166.667	2,221	166.667	5333	1.9	2666	0.953
Luffa-384	4,233	166.75	3,740	166.75	5336	1.26	1778	0.42
Luffa-512	4,593	166.667	3,700	166.75	5336	1.16	1777	0.58
Shabal	2,512	143.472	1,583	148.038	1469	0.584	367	0.146
SHAvite3-256	3,776	82.277	3,125	109.17	1170	0.309	1138	0.301
SHAvite3-512	11,443	63.666	9,775	59.4	931	0.081	918	0.08
SIMD-256	24,536	107.2	22,704	107.2	1338	0.054	1338	0.054
SIMD-512	44,673	107.2	43729	107.2	2677	0.059	2677	0.059
Skein-512	2,756	83.577	1,786	83.654	1945	0.706	973	0.353

5 Conclusions

In this paper we presented what we believe to be a methodology for fair and accurate comparisons of the SHA-3 hash functions. We implemented and tested all design variants to obtain full coverage of all of the hash functions as required by NIST. We developed a hardware wrapper to allow inclusion of padding and interfacing, to obtain the full timing and area analysis, and for completeness compared the area and speed of our hash designs both internal and external of this wrapper. Finally we presented throughput results for both long and short hash messages inclusive of this wrapper.

6 Acknowledgements

This material is based upon works supported by the Science Foundation Ireland under Grant No. 06/MI/006.

The support of the Informatics Commercialisation initiative of Enterprise Ireland is gratefully acknowledged.

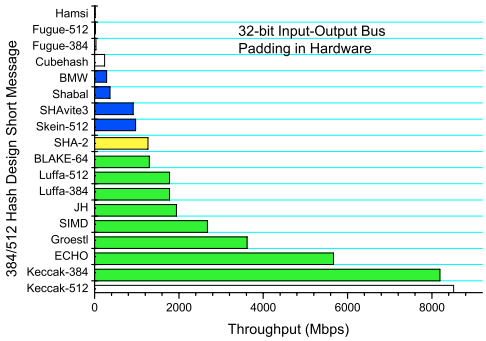

Figure 22: 384/512 Short Throughput

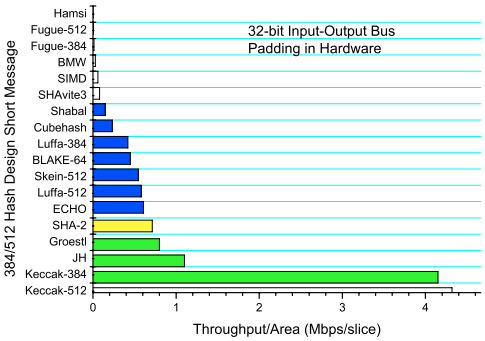

Figure 23: 384/512 Short Tp/Area

References

[1] NIST, "National institute of standards and technology. [docket no.: 070911510-7512-01] announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family," Federal Register, November 2007.

[2] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan, and W. P. Marnane, "Fpga implementations of sha-3 candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash," in *Digital Systems Design, Euromicro Symposium on*, 2009, pp. 783–790.

[3] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "A hardware wrapper for the SHA-3 hash algorithms," Cryptology ePrint Archive, Report 2010/124, 2010.

[4] E. Biham and O. Dunkelman, "A framework for iterative hash functions - HAIFA," Cryptology ePrint Archive, Report 2007/278, 2007.

[5] R. C. Merkle, "One way hash functions and DES," in *Advances in Cryptology — CRYPTO '89*, ser. LNCS, G. Brassard, Ed., vol. 435. Springer-Verlag, 1989, pp. 428–446.

[6] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology — CRYPTO '89*, ser. LNCS, G. Brassard, Ed., vol. 435. Springer-Verlag, 1989, pp. 416–427.

[7] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[8] E. Biham, R. J. Anderson, and L. R. Knudsen, "Serpent: A new block cipher proposal," in *Fast Software Encryption — FSE '98*, ser. LNCS, S. Vaudenay, Ed., vol. 1372. Springer-Verlag, 1998, pp. 222–238.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "On the indifferentiability of the Sponge Construction," pp. 181–197, 2008.

[10] R. S. Winternitz, "A secure one-way hash function built from DES," in *IEEE Symposium on Security and Privacy*. IEEE Press, 1984, pp. 88–90.

Appendix

As described in Section 2, we set the Input and Output bus, w, of our wrapper to 32-bits, a standard word size. Any hash function requiring a large message size, m, will be subject to a latency of m/w clock cycles. For these cases the clock count for the throughput needs to take this additional delay into consideration.

Also, while the wrapper itself does not affect the clock frequency, counters in the padding block may form the critical path and thus affect the timing, most notably for Haifa based designs. As such, padding in hardware may deplement some hash designs more than others.

In the interest of fairness we present further results to enable a more balanced review of the hash functions in hardware. These extra results feature our hash function implementations using an ideal bandwidth, where $w = m$, thereby negating any penalties incurred through transmission bottlenecks. We also present timing results for the padding in software both inclusive and exclusive of the Input and Output bus.

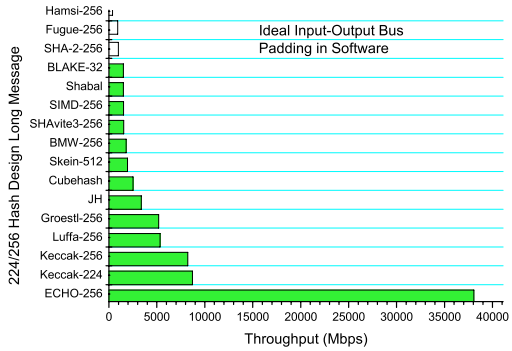

Figure 24: 224/256 Long Throughput

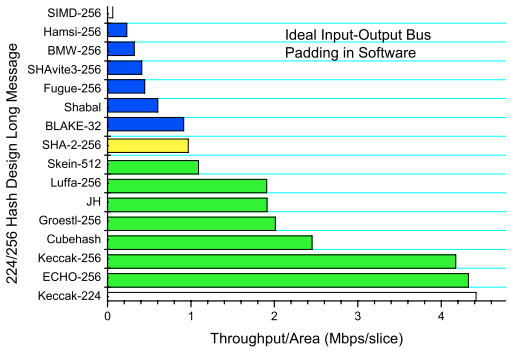

Figure 25: 224/256 Long Tp/Area

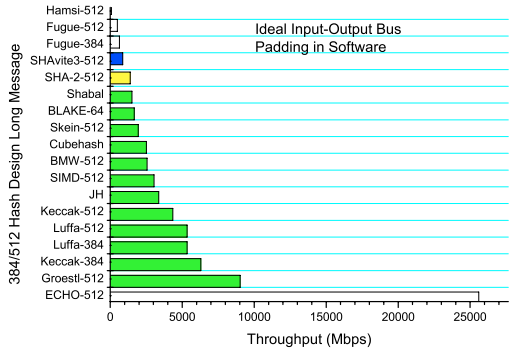

Figure 26: 384/512 Long Throughput

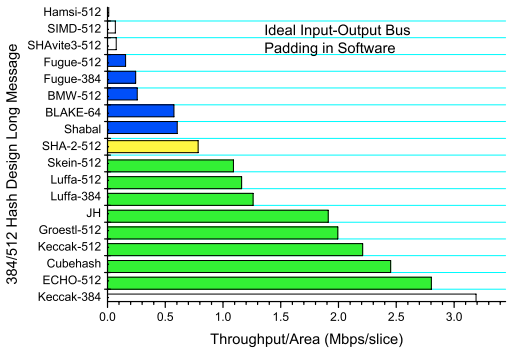

Figure 27: 384/512 Long Tp/Area

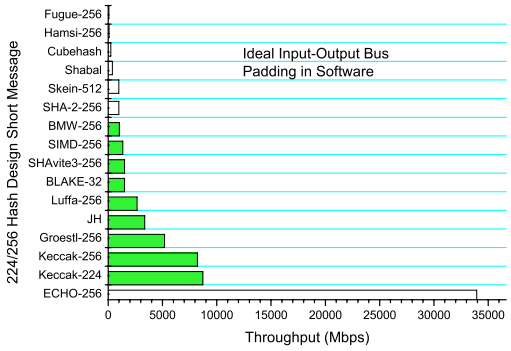

Figure 28: 224/256 Short Throughput

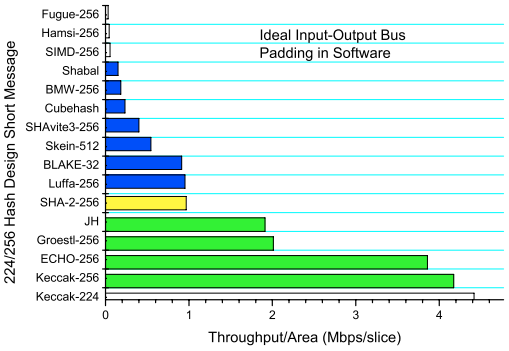

Figure 29: 224/256 Short Tp/Area

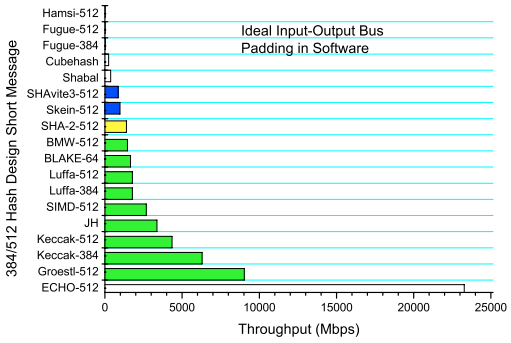

Figure 30: 384/512 Short Throughput

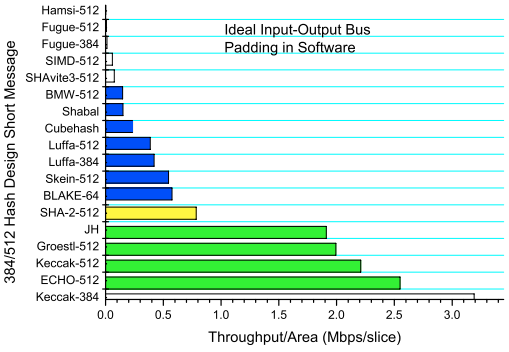

Figure 31: 384/512 Short Tp/Area

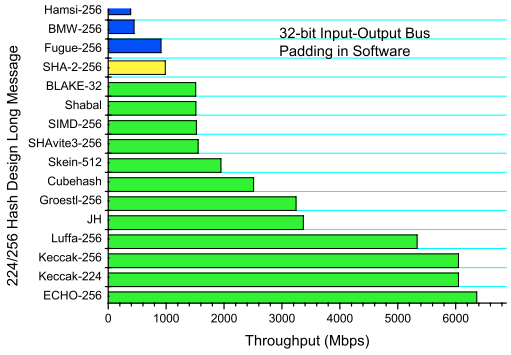

Figure 32: 224/256 Long Throughput

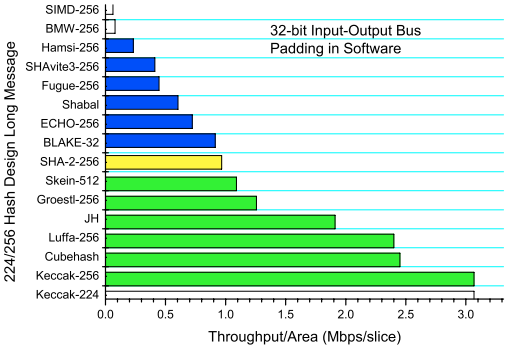

Figure 33: 224/256 Long Tp/Area

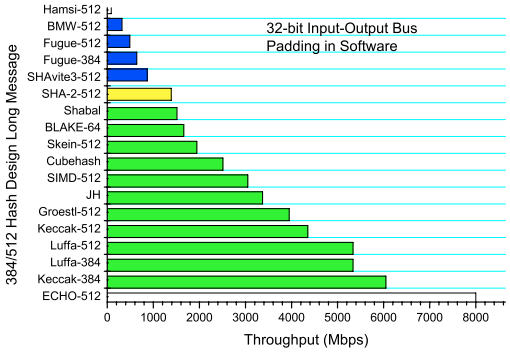

Figure 34: 384/512 Long Throughput

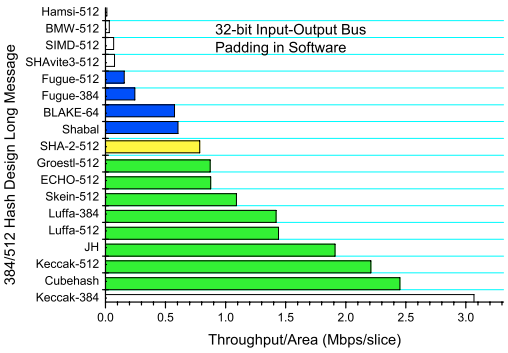

Figure 35: 384/512 Long Tp/Area

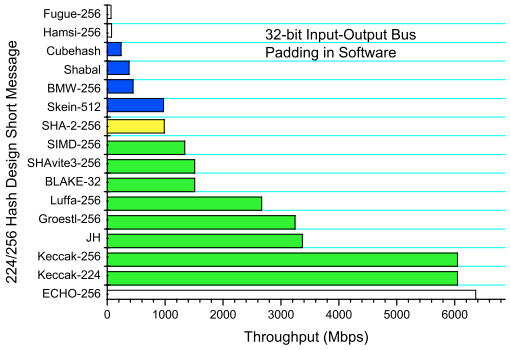

Figure 36: 224/256 Short Throughput

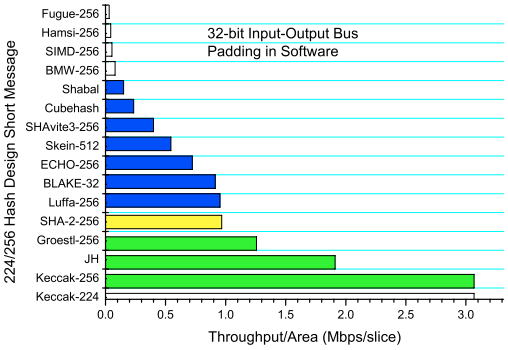

Figure 37: 224/256 Short Tp/Area

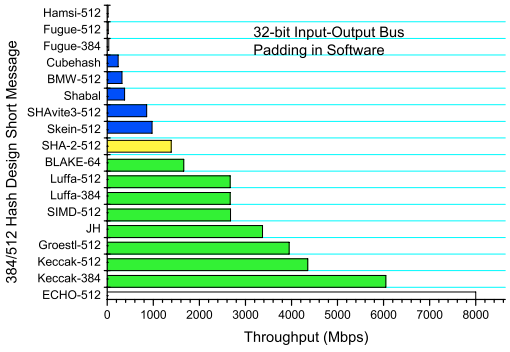

Figure 38: 384/512 Short Throughput

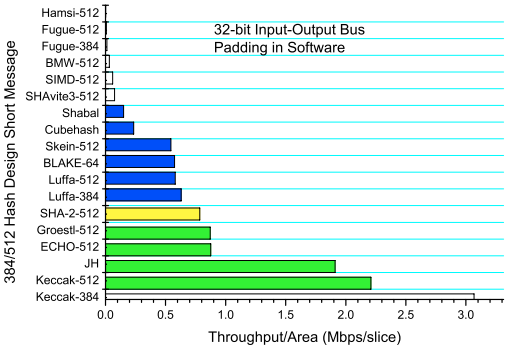

Figure 39: 384/512 Short Tp/Area