A Skein-512 Hardware Implementation

Jesse Walker
Intel Corporation
Security Research Lab
JF2-55
2111 NE 25th Ave
Hillsboro, OR USA 97124
jesse.walker@intel.com

Farhana Sheikh
Intel Corporation
Circuits Research Lab
JF2-04
2111 NE 25th Ave
Hillsboro, OR USA 97124
farhana.sheikh@intel.com

Sanu K. Mathew
Intel Corporation
Circuits Research Lab
JF2-04
2111 NE 25th Ave
Hillsboro, OR USA 97124
sanu.k.mathew@intel.com

Ram Krishnamurthy
Intel Corporation
Circuits Research Lab
JF2-04
2111 NE 25th Ave
Hillsboro, OR USA 97124
ram.krishnamurthy@intel.com

Abstract

This paper describes our Skein-512 hardware implementation. Skein is a semi-finalist in the NIST hash competition to create SHA-3, with Skein-512 being the primary submission. We compare our implementation of Skein-512 with other published hardware implementations of Skein, and with similar implementations for SHA-1 and SHA-2. We discuss four variations of our critical path to explore the throughput/latency tradeoffs afforded by the Skein algorithm, with the best tradeoff offering throughput of 58Gbps at a latency of 20 clock cycles.

1. Introduction

In 2005 Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu [23] unexpectedly discovered vulnerabilities in the widely used SHA-1 hash algorithm [17]. This attack called into question the practical security of SHA-1 when used in digital signatures and other applications requiring collision resistance. The SHA-2 family, which was designed to replace SHA-1, enjoys a similar structure, leading to concerns that it might as well fall to elaborations or variations of Wang's attack.

To respond to these concerns the United States government agency National Institute of Standards and Technology (NIST) is sponsoring an international competition to create a replacement algorithm, which will be called SHA-3 [18]. The competition drew 64 submissions, and to constitute a first round NIST identified 51 that met its minimum submission criterion. Based on public comments and internal reviews of the 51 first-round candidates, NIST later narrowed the field to a more manageable number of 14 semi-finalists, to enable deeper analysis.

The Skein hash algorithm [4] is one of the remaining 14 semi-finalists. This paper describes a new hardware implementation of Skein and explores implementation tradeoffs the algorithm enables. Section 2 touches on related work. Section 3 outlines the Skein algorithm. Section 4 describes our hardware implementation based on Intel's 32 nanometer process [16]. Section 5 examines performance issues and discusses tradeoffs to achieve implementations with problem-specific characteristics and compares our implementation with other known Skein implementations and with published hardware implementations of SHA-1, SHA-256, and SHA-512. Section 6 summarizes the paper and discusses future work.

2. Related Work

Numerous software implementations of Skein have been announced ([3], [5], [6], [7], [8], [15], [19], [22], [24]), but we are aware of few hardware implementations. This is not surprising for any of the SHA-3 candidates, given the newness of the algorithms.

To the best of our knowledge, M. Long developed the first Skein hardware implementation [11], based on a Xilinx FPGA. Long implemented Skein-256, but he scaled his results to Skein-512. Long implemented the main Threefish data path as a single 8 round pipeline. His design achieves a throughput of 871 Mbps. He reports a latency of 34 clocks. His design uses 586 flip flops (FF) the FPGA makes available, along with 7029 lookup tables (LUT) and 7508 LUT-FF pairs. He uses the latter as the metric for area cost.

In a paper that motivated our own work, S. Tillich et. al. from Graz University of Technology produced hardware implementations of all of the semi-finalist algorithms [21]. Their goal was to provide implementations using a common set of libraries, optimizations, system interfaces, and the like, to allow direct comparison of the algorithms. Their hardware implementation of Skein is poorly performing. We wanted to understand whether this was due to something intrinsic in the algorithm or whether they failed to capitalize on the opportunities the algorithm presents. Section 5 summarizes the Graz results in more detail and compares them with our Skein-512 design.

3. Skein Description

Skein is a family of algorithms based on one architecture, with block widths of 256, 512, and 1024 bits. Skein-512 is the primary submission, with Skein-256 defined for resource limited environments and Skein-1024 for high security applications. This paper focuses on Skein-512.

The Skein hash algorithm is composed of three components:

- Threefish, a wide-block tweakable block cipher,
- Unique Block Identification (UBI), a novel chaining mode, and
- An argument system to adapt Skein to different applications.

This section describes each of these, as well as how Skein assembles to form a hash algorithm. For a more complete description see the Skein specification.

3.1 Threefish

Skein is based on a family of tweakable block ciphers named Threefish. Each member of the Threefish family takes three parameters:

- An N-bit encryption key, where N is a power of 2 greater than or equal to 256,
- A 128 bit tweak, and
- An N bit block of plaintext to encrypt.

The Skein specification defines Threefish-N for 256, 512, or 1024. If E denotes the Threefish encryption function, K the key, T the tweak, and P a plaintext block, we write $E_{K,T}(P)$ for $E(K, T, P)$.

Threefish partitions its N-bit plaintext input into $w = N/64$ words, each of which is 64 bits. In each round the w words are grouped into $w/2$ pairs (A, B), each of which is input into a MIX function:

$$\text{MIX: } (A, B) \rightarrow (A+B, (B <<< R) \oplus (A+B)),$$

where "+" denotes 64-bit addition with carry, "$<<< R$" denotes left rotation by R bits, and "$\oplus$" XOR. The MIXes within a round can execute in parallel.

For Threefish-512, $w = 512/64 = 8$, meaning each round consists of 8/2 = 4 MIX functions. We name the MIX functions in a round MIX_0, MIX_1, MIX_2, and MIX_3 Threefish defines 8 rounds worth of rotation constants, repeated as many times as are needed; Threefish-512 uses each of the rotation constants 19 times for a total of 72 rounds. The rotation constants depend on N, and for Threefish-512 are

Round	MIX_0	MIX_1	MIX_2	MIX_3
0	38	30	50	53
1	48	20	43	31
2	34	14	15	27
3	26	12	58	7
4	33	49	8	42
5	39	27	41	14
6	29	26	11	9
7	33	51	39	35

Threefish-N permutes the words between rounds. The Threefish-N permutation re-pairs the w words, so that different words will mix in each round. The permutation is taken to have maximum period. The Threefish-512 permutation is

Input word number	0	1	2	3	4	5	6	7
Output word number	2	1	4	7	6	5	0	3

Threefish-N adds a round key $K_r = K_{r,0}\, K_{r,1}\, \ldots\, K_{r,w-1}$ to the words before the 0th round and then after every fourth round r, where each K_{ri} is a 64-bit word. The round key is computed from the Threefish key $K = K_0\, K_1\, \ldots\, K_{w-1}$ and the tweak $T = T_0\, T_1$ as follows: set

$$K_w = K_0 \oplus K_1 \oplus \ldots \oplus K_{w-1} \oplus \lfloor 2^{64}/3 \rfloor$$

$$T_2 = T_0 \oplus T_1$$

Then

$$K_{s,i} = K_{(s+i) \bmod (w+1)} \qquad \text{for } i = 0, 1, \ldots, w-4$$

$$K_{s,i} = K_{(s+i) \bmod (w+1)} + T_{s \bmod 3} \qquad \text{for } i = w-3$$

$$K_{s,i} = K_{(s+i) \bmod (w+1)} + T_{(s+1) \bmod 3} \qquad \text{for } i = w-2$$

$$K_{s,i} = K_{(s+i) \bmod (w+1)} + s \qquad \text{for } i = w-1$$

where s = $\lfloor r/4 \rfloor$ denotes the subkey number and i the word number.

3.2 UBI

Skein replaces the Davies-Meyer, Merkle-Damgård, and Merkle-Damgård strengthening constructions in classical hash function designs with UBI. The input to UBI consists of three items:

- An initialization vector IV,
- The string M to hash. M can consist of up to $2^{99} - 8$ bits in length, and
- An application-specific $Type$, which separates different uses.

Thus, $UBI(IV, M, Type)$ is the signature for a UBI operation.

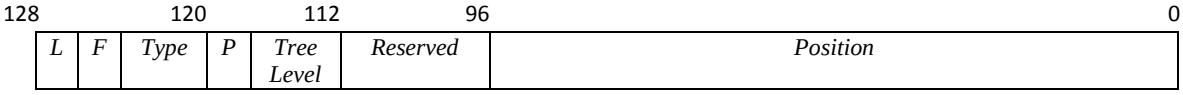
The steps to process M under UBI are

- Initialize the initial chaining variable value H_0 to the initialization vector value IV.
- Parse M into N-bit blocks $M_1, \ldots, M_m$, where N is the block size of an underlying tweakable block cipher E. 0-pad the final message block M_m if its length is not already a multiple of N bits.
- For each message block M_i:
    - Compute the block cipher tweak T_i for each message block M_i using the block offset and the $Type$ as described below
    - Compute the next chaining variable by applying the block cipher E in Matyas-Meyer-Oseas mode to the next block M_i, its tweak T_i, and the previous chaining variable value:

$$H_i = E_{H_{i-1},T_i}(M_i) \oplus M_i.$$

- Output the final chaining variable value H_m

The tweak is constructed as



where

- L (bit 127) = 1 for last block M_m and 0 otherwise
- F (bit 126) = 1 for first block M_1 and 0 otherwise
- $Type$ (bits 120-125) = the application-specific UBI function being performed
- P (bit 119) = 1 if the message block is padded and 0 otherwise
- $Tree\ Level$ (bits 112-118) = level of the tree when tree hashing is used; 0 for non-tree computations
- $Reserved$ (bits 96-117) = for future use; must be 0
- $Position$ (bits 0-95) = number of bytes of M processed so far

In Skein-N, UBI always uses Threefish-N as its tweakable block cipher

3.3 Skein Argument System

The Skein argument system specifies the UBI Type value. The types are

- *Key* Value = 0 information hashed is a key (for MACs and KDFs)
- *Cfg* Value = 4 for computing the configuration block (initialization vector)
- *Prs* Value = 8 for personalized hashing
- *PK* Value = 12 personalization using a public key
- *Kdf* Value = 16 key identifier
- *Non* Value = 20 nonce
- *Msg* Value = 48 message that Skein is hashing
- *Out* Value = 63 for computing Skein output

3.4 Putting the Pieces Together

The Skein hash of a message M uses three UBI calls. The first constructs the initialization vector for hashing message M:

$$IV = UBI(0^N, Config, Cfg)$$

where *Config* is a special 32 byte string to configure the IV. The following table describes each of the bytes of *Config*:

Offset in bytes from *Config* string start	**Size in Bytes**	**Name**	**Description**
0	4	Schema	0x53 0x48 0x41 0x33 (the ASCII string "SHA3")
4	2	Version number	1, encoded as a 16 bit integer
6	2	Reserved	Set to 0
8	8	*Outbits*	Number of output bits, encoded as a 64 bit integer
16	1	Tree leaf size	Set to 0 if not use
17	1	Tree fanout	Set to 0 if not used
18	1	Tree height	Set to 0 if not used
19	13	Reserved	Set to 0

The second UBI invocation uses the initialization vector to hash of the input message M.

$$G = UBI(IV, M, Msg)$$

The final UBI call expands the intermediate value G into the desired number of output bytes $o = \lceil Outbits/8 \rceil$. Here

$$Output = Truncate(UBI(G, 0, Out) \,||\, UBI(G, 1, Out) \,||\, \ldots \,||\, UBI(G, o, Out), o)$$

where each of the values $0, 1, \ldots, o$ is encoded as a 64 bit integer, $a \,||\, b$ denotes concatenation of strings a and b, and $Truncate(a, b)$ truncates string a to b bytes. To meet NIST's requirements the primary submission restricts $Outbits$ to 128, 160, 224, 256, 384, and 512, so $o = 1$ and this simplifies to

$$Output = Truncate(UBI(G, 0, Out), o).$$

4. Hardware Implementation

We implemented Skein-512 on a 32nm CMOS process, at 0.80V and 110°C [16]. The core data path consists of eight unrolled rounds of Threefish which are pipelined. This allows parallel computation of two independent hashes, but requires two tweak generators and two key schedulers to independently supply two subkeys to keep the hardware pipeline filled with two independent messages during each cycle. Each pipeline stage encompasses an addition of the 512b subkey with the 512b message and four 512b Threefish Mix and Permute rounds.

This pipelined 8-round Threefish implementation can hash two input messages in parallel at the expense of increased latency, from 10 cycles to 20 cycles. An eight-round implementation allows for fixed rotation distances for each Mix block enabling a simplified, compact implementation with a total area of 60.4K gate equivalents. Even greater parallelism is possible by inserting registers into the pipeline at every one or two rounds, but each such addition doubles the latency and increases area. An overview of the hardware implementation is given in Figure 1 below.

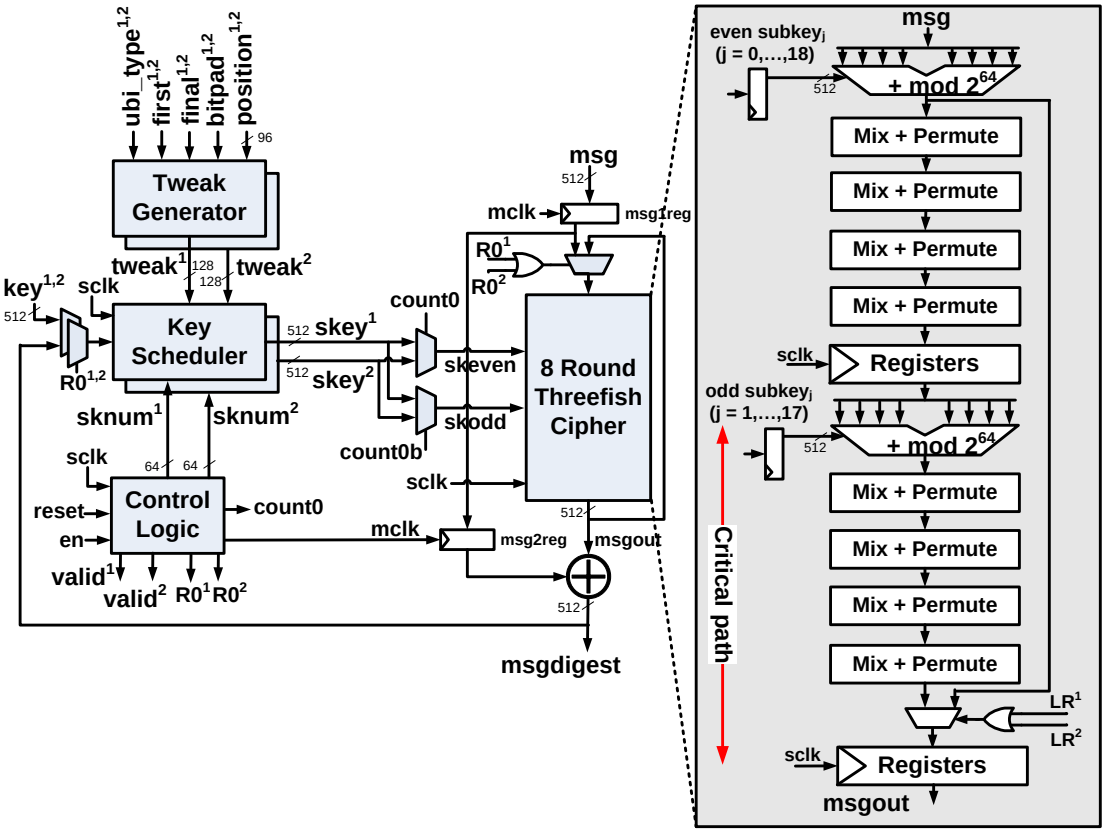


Figure 1: Skein-512 hardware implementation

4.1 Threefish Cipher Implementation

The 8-round Threefish data path depicted in Figure 1 consists of 32 Mix blocks, 8 Permute blocks, and 16 64b modulo-2^{64} adders. Figure 2 below shows the details for four rounds. Each Mix block consists of a 64b modulo-2^{64} adder followed by an XOR of the adder output with a rotated version of second adder input. Skein-512 requires a total of 32 rotation constants, which we hard-code into each of the 32 Mix blocks. Each round consists of four parallel 64b x 64b Mix blocks followed by a permutation of the eight 64b words of the Mix block outputs. The permutations are accomplished via signal routing as shown in Figure 2. After every four rounds, eight parallel 64b modulo-2^{64} adders compute the addition of the 512b input word with a new 512b subkey generated using a 128b tweak and 512b key. Four Threefish rounds and one addition of the input word with the subkey are computed during each main clock cycle, which Figure 2 names as sclk.

The modulo-2^{64} adders are the key components of the critical path. They are implemented using Intel's standard cell library and sizing is optimized for minimum delay through the critical path. A final addition of the 512b data input and final subkey follows 72 rounds of Mix and Permute and is output directly as the Threefish cipher. The output of the second stage of registers is controlled by signals LR1 and LR2 that are asserted after 19 and 20 cycles, respectively. If LR1 (last round for message 1) or LR2 (last round for message 2) are high, then output of the addition after 72 rounds is fed directly to the registers, otherwise the output of the Permute block is fed to the second set of the registers.

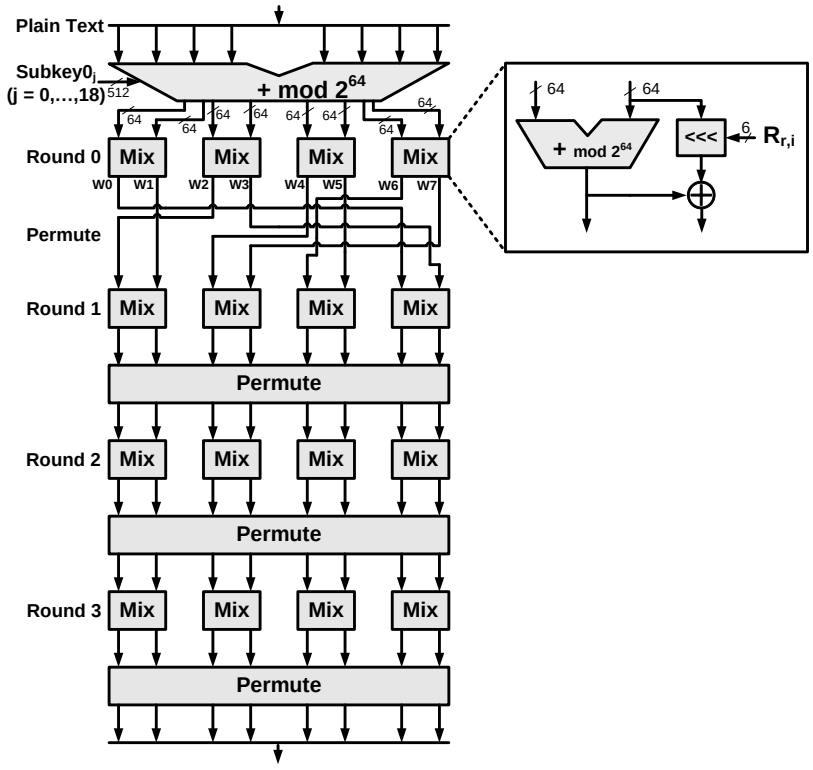


Figure 2: Four rounds of the pipelined 8-round Threefish cipher datapath

4.2 Key Scheduler Implementation

At the start of each UBI invocation, each Tweak Generator constructs a 128b tweak that is fed into each Key Scheduler that computes the 512b subkey based on the 512b input key and 128b tweak. The Key Scheduler operates using the main clock, sclk. During each sclk cycle, each Key Scheduler creates the next 512b subkey, which it provides to the eight-round Threefish cipher block. Figure 2 shows the Key Scheduler in detail. In the initial round, the first five 64b subkey words (skey0 – skey4) are equivalent to the five input 64b key words (k0 – k4). The sixth subkey word, skey5, is the result of adding the first 64b tweak word, t0, with k5. Subkey word skey6 is the result of the addition of tweak t1 with k6. The last subkey word, skey7, is the output of the addition of the subkey number with k7. Key word, k8, is the result of the XOR of all the key words, k0 – k7 and the constant shown in Figure 2. Tweak, t2, is the result of XORing t0 and t1. In subsequent rounds, the key words are rotated as well as the tweaks as shown in Figure 2.

The Tweak Generator that feeds the Key Scheduler is a simple block that takes ubi_type, first, final, bitpad flags, and the 96b position vector to build the 128b tweak as described in [4].

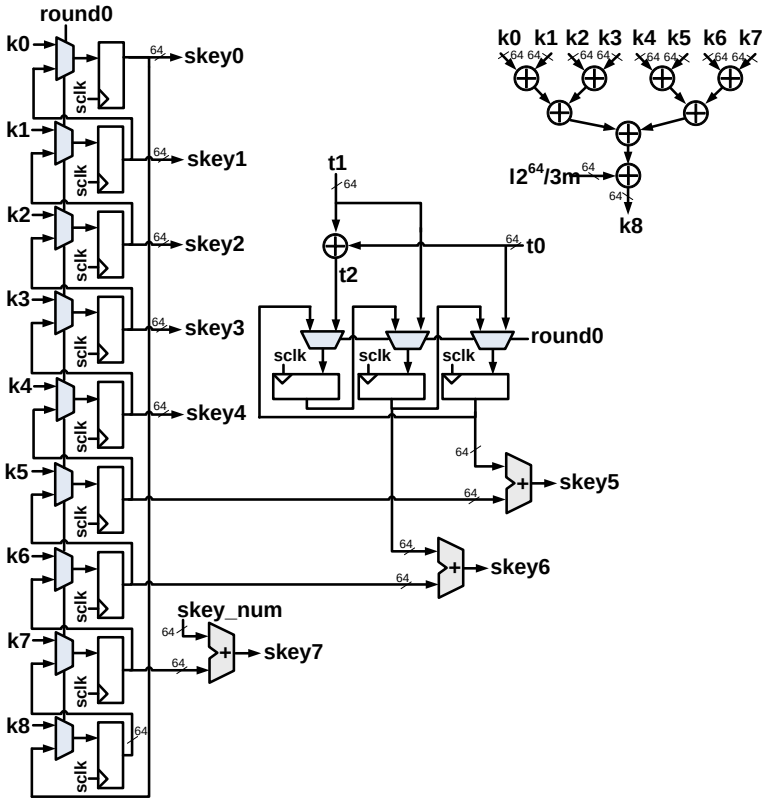


Figure 3: Key Scheduler Implementation

4.3 Control Logic and Timing

Sclk is the main clock in the design. The design uses sclk to clock the pipelined 8-round Threefish cipher datapath, the Key Scheduling circuit, and the 5b counter, which is the design's main control mechanism. The 5b counter tracks the round number and is reset every 20 sclk cycles. Two registers control the input messages to be hashed. These registers are clocked using a secondary clock, mclk. Mclk is clocked at the same frequency as sclk for two cycles but then falls silent for the next 18 cycles, as depicted in Figure 4 below. During the first sclk cycle, cycle 0, m1 is latched into the first register, msg1reg, and fed to the 8-round Threefish cipher datapath, where it is added to the incoming subkey. In the second sclk cycle, m1 moves from the first message register into the second one, msg2reg, where it is held for 72 rounds (19 cycles) and so that it can be XORed with output of the Threefish cipher. When the first message is latched into msg2reg, a second message m2 is latched into msg1reg, so that processing can start on it in the second sclk cycle, cycle 1. As can be seen from the timing diagram in Figure 4, m1 and m2 are processed simultaneously, with m1 processing completing one cycle earlier than m2.

A new subkey is delivered to the 8-round Threefish cipher every cycle from both key schedulers. Even cycle subkeys are fed to the first set of 8 parallel modulo-2^{64} adders and odd cycle subkeys are fed to the second set of 8 parallel modulo-2^{64} adders (Figure 1). When count[0] is high, the even subkey (skey1) for m1 is used in the first set of adders and the second set of adders is fed the odd subkey (skey2) for m2. When count[0] is low, the even skey2 for m2 is used in the first set of adders and the odd skey1 for m1 is used in the second set of adders. This multiplexing is controlled by the least significant digit of the 5b

counter, count[0] (i.e. count0 is high when count[0] is high and count0b is high when count[0] is low). The current round number for m1 and m2 processing is fed to the two key schedulers every cycle and the R01 and R02 signals that control the output of the multiplexer to the 8-round Threefish cipher datapath are high when the round count is zero. Valid signals (valid1 and valid2) are asserted when the count reaches 20 for m1 and 21 for m2, signifying that the message digest is valid.

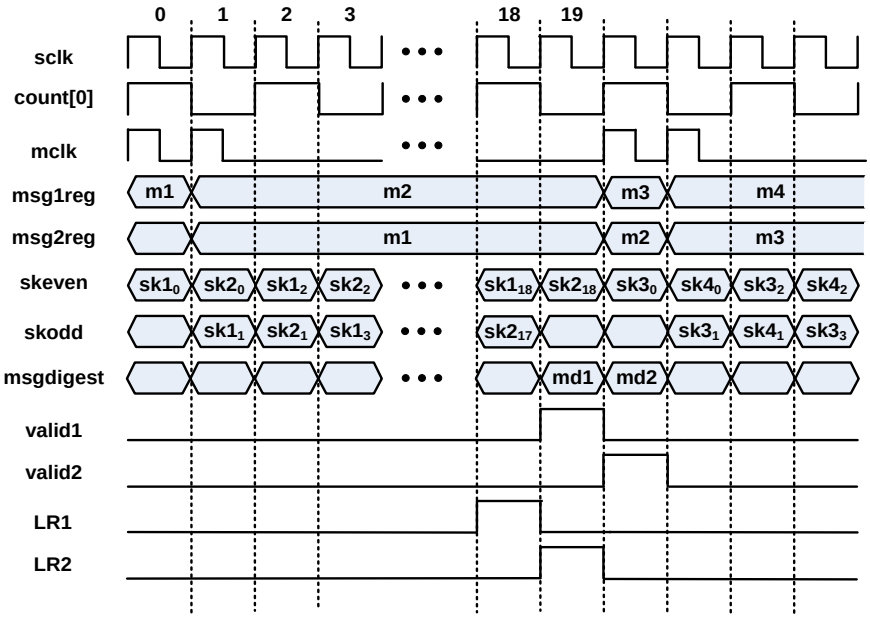


Figure 4: Skein SHA-3 hardware accelerator timing

5. Area, Latency and Throughput Tradeoffs

We designed the 8-round Threefish-512 cipher data path in four different ways to better understand the area, latency, and throughput tradeoffs possible with Skein. The first version, iSkein-512-8R-1M is the 8-round data path without any pipelining. An implementation based on this design hashes one message every 10 cycles with a throughput of 32Gbps, at 631MHz clock frequency. Our second implementation, iSkein-512-4R-2M, is pipelined once so that four rounds execute every sclk cycle as described in the previous section. With this implementation, the latency is increased to 20 cycles at 1.13GHz clock frequency but enables parallel processing of two independent messages. The throughput improves by 45% to 58Gbps. Our third implementation, iSkein-512-2R-4M, pipelines the design further so that two rounds are processed every sclk cycle resulting in total latency of 40 cycles with clock frequency set to 1.74GHz. This allows hashing of four independent messages in parallel, improving throughput to 89Gbps. If the eight rounds are fully pipelined (iSkein-512-1R-8M), then it is possible to hash eight messages in parallel with 80-cycle latency, resulting in throughput of 122Gbps at 2.38GHz.

Designing in support for hashing simultaneous parallel messages is not unreasonable. [4] reports that Skein-512 achieves excellent performance in software, so a hardware implementation is most likely to be used at a system chokepoint such as a router, traffic-shaper, or the like, where message load can overwhelm a processor due to multiple simultaneous input streams. We believe our choice of two

parallel hashes is the best tradeoff for most hardware implementations. In terms of area, as we continue to add pipeline stages, the area increases due to the additional registers required. The iSkein-8R-1M implementation that has 10-cycle latency and runs at the slowest clock frequency, has a total area of 57.9K gate equivalents. The once pipelined version, iSkein-4R-2M, consumes 60.4K gates and has a latency of 20 cycles at 1.13GHz. The third version, iSkein-2R-4M, that increases the number of pipeline stages to four, and latency to 40 cycles, is implemented in 63K gates. The version with shortest clock cycle, 2.38GHz but with 80-cycle latency, consumes the largest area at 70.1K gate equivalents.

Since Skein is a relatively young algorithm, few synthesized hardware implementations appear in the literature. In this section we compare our Skein-512 implementation with that produced at Graz University [21]. To get a better context for this comparison, we also compare our implementation with the best performing SHA-1 and SHA-2 implementations of which we are aware from the literature.

5.1 Comparison with the Graz Skein-512-512 Implementation

In [21] S.Tillich and his Graz University colleagues report a Skein-512 design. They targeted a 180nm process using a standard cell library produced by Faraday. Their goal was to build implementations of each of the semi-finalist algorithms, optimized for maximum throughput, to create a fair comparison among algorithms. Since our design uses a 32nm process and different standard cell libraries, the results cannot be compared with complete fairness. However, to get an understanding of the differences between the two designs we have scaled the results reported in [21] to our environment using a direct scaling methodology. We have allowed a 20% improvement in performance for each technology generation, as we scale from 180nm to 32nm. This allows us to compare our work with published work in [21] at the same technology node, thereby giving some insight into the relative performance of our eight-round iSkein-512-8R-1M design compared to the Skein-512-512 design by Graz University and their best performing implementation. It also highlights the benefits of pipelining and the relative area/throughput/latency tradeoffs.

Based on these assumptions, the results discussed above are summarized below in Table 1 which also includes the results from Graz University [21] scaled to 32nm CMOS process technology. Figure 5 below shows throughput vs. latency tradeoffs for our design and the expected performance of the Graz design if implemented in a 32nm CMOS process. Our iSkein-4R-2M design achieves 9% performance improvement and 20% latency improvement as compared to the best performing scaled design by the Graz University team, their Keccak implementation.

The area and latency tradeoffs are shown in Figure 6 above for all designs scaled to 32nm. As the latency of the design increases, the area also increases due to the additional registers. However, these additional pipeline stages enable hashing of multiple independent messages in parallel, thereby improving throughput. The cost of improved throughput is higher area. The area of iSkein-4R-2M is 6.7% higher than the scaled SHA-3 Keccak implementation by Graz University (see also Table 1).

Table 1: Results for implementation in 32nm CMOS technology

Implementation	Block (bits)	Clock Frequency (MHz)	Latency (cycles)	Area (Gate Equivalents)	Throughput (Gbps)
Intel Technology Implementations					
iSkein-512-1R-8M	512	2380.95	80	70,071	121.90
iSkein-512-2R-4M	512	1736.11	40	62,954	88.89
iSkein-512-4R-2M	512	1126.13	20	60,395	57.66
iSkein-512-8R-1M	512	631.31	10	57,931	32.32
Graz University Implementations Scaled to 32nm					
BLAKE-32	512	424.61	22	45,640	9.88
BMW-256	512	408.58	53	122,092	3.95
CubeHash16/32-h	256	362.72	8	58,872	11.61
ECHO-256	1536	352.94	97	141,489	5.59
Fugue-256	32	636.39	2	46,257	10.18
Grostl-256	512	672.52	22	58,402	15.65
Hamsi-256	32	432.74	1	58,661	13.85
JH-256	512	946.11	39	58,832	12.42
Keccak (256)	1088	1213.80	25	56,316	52.82
Luffa-224/256	256	1202.08	9	44,972	34.19
Shabal-256	512	797.53	50	54,186	8.17
SHAvite-3_256	512	220.39	19	58,828	5.94
SIMD-256	512	161.57	36	104,166	2.30
Skein-256-256	256	182.94	10	58,611	4.68
Skein-512-512	512	121.60	10	102,039	6.23

iSkein-4R-2M cannot be compared directly with the Graz University design Skein-512-512, because ours achieves its throughput by hashing two messages in parallel. iSkein-8R-1M is directly comparable. The iSkein-8R-1M implementation achieves a 5X throughput improvement over the Graz University Skein-512-512 implementation scaled to 32nm. This difference in performance can be due to many factors, such as different 64b adder implementations, differently performing standard cell library, the availability of high performing standard cells, and possibly a different critical path. Figure 1 highlights our critical path. The critical path travels through a series of five modulo-2^{64} 64b adders. Our implementation uses high-speed adder implementations whose delay is O(log(N)), where N is the number of bits. These adders are part of our standard cell library. Slower adders, such as ripple-carry style, increase critical path delays due to the carry propagation at each bit. Another advantage of our design is that our control mechanism relies solely on some simple logic and a 5b counter. We have also eliminated additional memory overhead by choosing not to pre-compute and store the output of the first UBI invocation as has been done in the Graz University implementation. It is our belief, that by optimizing the critical path and carefully constructing the control, the Skein SHA-3 candidate can operate at high clock frequencies, above 1GHz, resulting in throughputs comparable to the best performing Graz implementation.

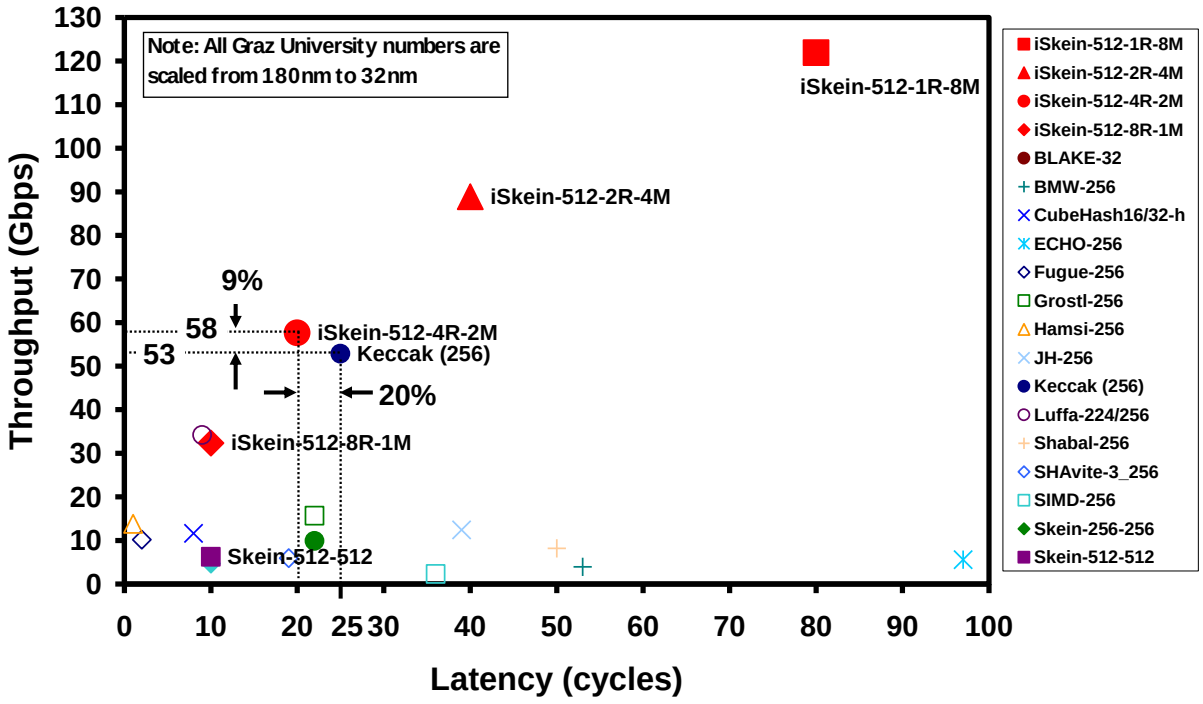


Figure 5: SHA-3 circuits – estimated throughput vs. latency (32nm CMOS)

The red iSkein-512-* points represent performance of the Intel designs. The other points represent the Graz results reported in [21][20] scaled to 32 nm

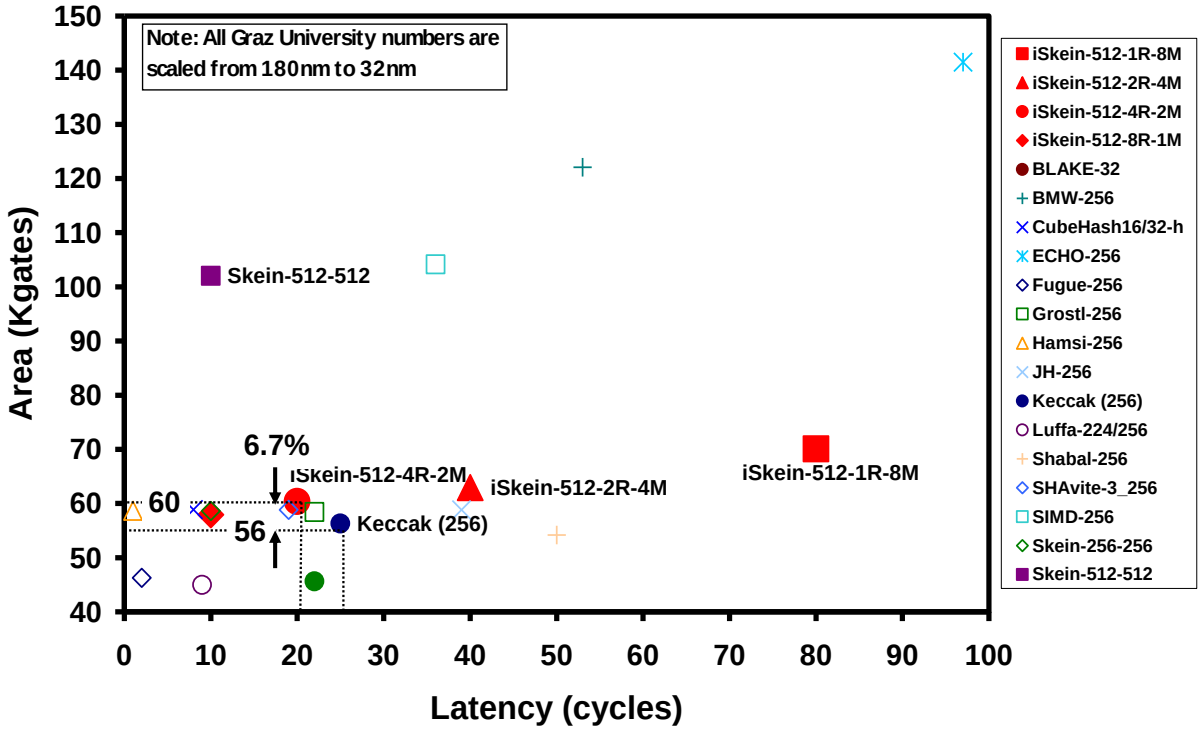


Figure 6: SHA-3 circuits - estimated area vs. latency (32nm CMOS)

5.2 Comparison with SHA-1 and SHA-2 Implementations

We compare our Skein-512 implementation to those for SHA-1 and SHA-2 scaled to the 32 nm process. Figures 6 and 7 provide block diagrams depicting the salient features of these designs.

The SHA-1 message digest data path operates on 32b operands, with hash values stored in registers A-E (Fig. 7). The critical 'A'-computation involves a 5b left-rotate, an iteration-dependent compression function ft and five-way 32b addition. These computations iterate for 80 rounds after which A-E values are concatenated to obtain the 160b hash.

$$f_t = \begin{cases} \text{Ch(B,C,D)} = BC \oplus \bar{B}D, & 0 \le t \le 19 \\ \text{Parity(B,C,D)} = B \oplus C \oplus D, & 20 \le t \le 39 \\ \text{Maj(B,C,D)} = BC \oplus BD \oplus CD, & 40 \le t \le 59 \\ \text{Parity(B,C,D)} = B \oplus C \oplus D, & 60 \le t \le 79 \end{cases}$$

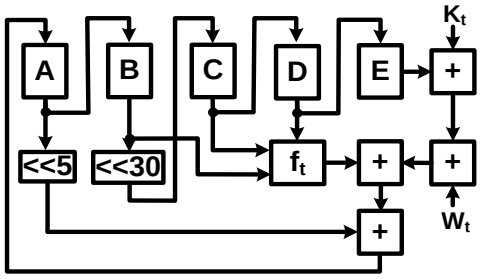


Figure 7: SHA-1 message digest datapath

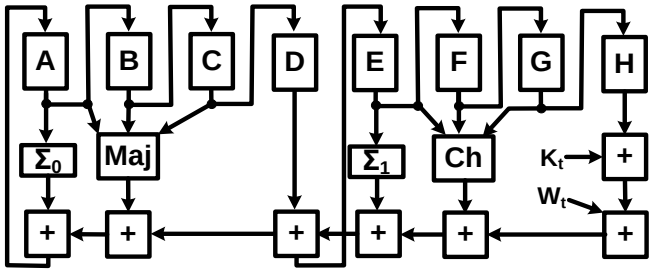


Figure 8: SHA-2 message digest datapath

Message digest computation for the SHA-2 family of hashes uses 32 datapath slices for SHA-224/256 and 64 datapath slices for SHA-384/512 (Figure 8). Hash values are stored in registers A-H with 64 and 80 iterations for SHA-224/256 and SHA-384/512 respectively. SHA-2 hashes use more complex diffused rotations $\sum_0$ and $\sum_1$ compared to the left rotates used in SHA-1:

$$\text{SHA-224/256: } \Sigma_0(x) = \text{ROTR}^2 \oplus \text{ROTR}^{13} \oplus \text{ROTR}^{22}$$
$$\text{SHA-384/512: } \Sigma_0(x) = \text{ROTR}^{28} \oplus \text{ROTR}^{34} \oplus \text{ROTR}^{39}$$
$$\text{SHA-224/256: } \Sigma_1(x) = \text{ROTR}^6 \oplus \text{ROTR}^{11} \oplus \text{ROTR}^{25}$$
$$\text{SHA-384/512: } \Sigma_1(x) = \text{ROTR}^{14} \oplus \text{ROTR}^{18} \oplus \text{ROTR}^{41}$$

The critical 'A' computation involves 7-way addition, 3-way rotated diffusion and 3-input compression functions.

Several hardware techniques have been used to improve SHA performance ([2], [9], [12], [13], [14], [20]). These include carry save addition of intermediate results with compressors ([2], [12], [20]), loop unrolling to mitigate the serial dependence of hash computation [9] and pipelining [20] to improve throughput.

To the best of our knowledge, the fastest reported SHA accelerator measurements, ported to 32nm technology achieves 6.6 Gbps SHA-1 throughput [1]. Our iSkein-4R-2M implementation achieves a

throughput of 9X over this. Similarly, the best published SHA-2 implementation achieves throughput of 18Gbps with a latency of 84 cycles [10]. Our iSkein-4R-2M exceeds the throughput of this by over 3X with a quarter of the latency.

6. Summary

This paper describes the design of a hardware implementation of Skein-512, a candidate algorithm being considered for SHA-3. On a 32nm process, the design achieves a throughput of 58Gpbs with a latency of 20 clock cycles.

Our data suggests that our iSkein-4R-2M hardware implementation is among the best implementation choices for the SHA-3 candidates. The iSkein-4R-2M design shows a 9% performance improvement and 20% latency improvement over the best performing scaled Graz implementation. We therefore believe our work shows that Skein can achieve very good performance when implemented in hardware.

Many research topics remain for future work. Our implementation does not support Skein-256 and Skein-1024, nor does it support any of Skein's optional arguments such as personalization, KDF, and MAC. We would like to evolve our design to add this support. Finally, once NIST announces the SHA-3 finalist algorithms, we hope to create hardware designs for each.

References

[1]. D. Carlson, D. Brasili, A. Hughes, A Jain, T. Kiszely, P. Kodandapani, A. Vardharajan, T. Xanthopoulos, V. Yalala, "A High Performance SSL IPSEC Protocol Aware Security Processor", *ISSCC Digest of Technical Papers*, pp. 142-143, Feb. 2003.

[2]. R. Chaves, G. Kuzmanov, L. Sousa, S. Vassiliadis, "Cost-Efficient SHA Hardware Accelerators", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 16, NO. 8, pp 999-1008, August 2008.

[3]. Fajarado, A., "Skeinfish," http://code.google.com/p/skeinfish/downloads/list/SkeinFish-0.4.1.zip

[4]. Ferguson, N., S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker, "The Skein Hash Function Family, Version 1.1," November 15, 2008, http://www.skein-hash.info/sites/default/files/skein1.1.pdf

[5]. Finne, S., "A Cryptol implementation of Skein," http://www.galois.com/blog/2009/01/23/a-cryptol-implementation-of-skein/

[6]. Fürstenau, H, "Skein extension module for Python 3.0," http://packages.python.org/pyskein/

[7]. Kausche, M., "Skein Implementation in Ada," http://www.skein-hash.info/node?page=1/

[8]. Krishnan, S., "nskein – a Skein implementation in .NET," http://github.com/sriramk/nskein

[9]. Y. Lee, H. Chan, I. Verbauwhede, "Throughput Optimized SHA-1 Architecture Using Unfolding Transformation", *Application-specific Systems, Architectures and Processors (ASAP'06)*, pp. 354 - 359, 2006.

[10]. Y. Lee, H. Chan, I. Verbauwhede, "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations", *8th Workshop for Information Security (WISA'07)*, pp. 102 - 114, 2007.

[11]. Long, M., “Implementing Skein Hash Function on Xilinx Virtex-5 FPGA Platform, February 2, 2009, http://www.skein-hash.info/sites/default/files/skein_fpga.pdf

[12]. M. Macchetti, L. Dadda, "Quasi-Pipelined Hash Circuits", *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pp. 222-229, 2005.

[13]. R. McEvoy, F. Crowe, C. Murphy, W. Marnane, "Optimisation of the SHA-2 Family of Hash Functions on FPGAs", *Proceedings of the 2006 Emerging VLSI Tech. and Arch.* (*ISVLSI'06),* pp. 317-322, *200*6.

[14]. M. McLoone and J. V. McCanny, “Efficient single-chip implementation of SHA-384&SHA-512”, *IEEE International Conference on Field-Programmable Technology*, pp. 311–314, 2002.

[15]. Mueller, T., “Skein Implementation in C#”, http://www.hotpixel.net/software.html#skein512.net

[16]. Natarajan, S., et. al., “A 32nm Logic Featuring 2nd Generation High-K + Metal Gate Transistors, Enhanced Channel Strain and 0.171µm2 SRAM Cell Size in a 291Mb Array,” IEDM Tech. Dig., paper 27.9, Dec. 20

[17]. NIST, “FIPS PUB 180-3 Secure Hash Standard (SHS),” October 2008, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[18]. NIST, “Announcing Request for Candidate Nominations for a new Cryptographic Hash Algorithm (SHA-3) Family,” Federal Registry Vol 72, No 212, November 2, 2007

[19]. Otte, D., “Skein Implementation for AVR Microcontroller,” http://www.das-labor.org/wiki/AVR-Crypto-Lib/en

[20]. Satoh, T. Inoue, "ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS", *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, Vol. 1, pp. 532-537, 2005.

[21]. Tillich, S., M. Feldhofer, M. Kirschbaum, T. Plos, J.M. Schmidt, and A. Szekely, “High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein,” Graz University of Technology, October 21, 2009

[22]. Walter, J., 8-bit Implementation of Skein, http://www.syntax-k.de/projekte/fhreefish/fhreefish-1.2.1.zip

[23]. Wang, X., Y. L. Yin, and H. Yu, Collision Search Attacks on SHA1,” February 13, 2005, http://www.c4i.org/erehwon/shanote.pdf

[24]. Whiting, D., “Skein Source Code and Test Vectors,” http://www.skein-hash.info/downloads/skein_NIST_CD_121508.zip