Optimization failures in SHA-3 software

Daniel J. Bernstein

Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

1 Introduction

This paper analyzes the software speeds of the SHA-3 finalists on various CPU microarchitectures. The analysis compares

- upper bounds for the cycles used by each hash function on each microarchitecture—the speeds reported for the best implementations available in eBASH [1]—to
- lower bounds for the cycles used by the same implementation strategies.

In some cases the lower bounds and upper bounds are close together, demonstrating that optimization work has come nearly to an end (unless someone comes up with a better implementation strategy). However, in many more cases there are large gaps between the lower bounds and the upper bounds. Obviously these gaps are caused by a combination of

- a lack of tightness in the lower bounds—a failure of the lower bounds to account for some cycles that must be spent—and
- a lack of tightness in the upper bounds—a failure of the implementations to eliminate some cycles that need not be spent.

The second effect is more important for SHA-3 users, and is highlighted in the title of this paper. I expect continued work to reduce these gaps, both by improving the lower bounds and improving the implementations. Implementors interested in improving SHA-3 software should feel free to contact me to coordinate further work.

2 Review of ALUs and other bottlenecks

The most important resources in a microarchitecture are its arithmetic-logic units (ALUs). For example, each core on an x86 Atom CPU or armeabi Cortex A CPU has two 32-bit ALUs; each core on an amd64 Atom or Bobcat CPU has two 64-bit ALUs; and each core on a high-power Intel/AMD CPU has three 64-bit ALUs. Each b-bit ALU follows one b-bit instruction per cycle, such as an instruction that subtracts two b-bit integers modulo 2^b. One might guess that a computation involving N 64-bit arithmetic operations is equivalent to a computation involving $2N$ 32-bit operations, and thus runs in

- $N/3$ cycles on amd64 Sandy Bridge, amd64 K10, etc.;
- $N/2$ cycles on amd64 Atom or amd64 Bobcat; and

This work was supported by NIST grant 60NANB10D263. Permanent ID of this document: 5b2ffa1851349170612efe8dc33f8158. Date: 2012.01.04.

– N cycles on x86 Atom or armeabi Cortex A.

However, computations are often faster than this, for several reasons:

– Most microarchitectures include vector ALUs that use extra circuitry to apply the same arithmetic instruction to two or more operands in parallel. All of the Intel and AMD microarchitectures include 128-bit vector instructions, as do the ppc32 G4 and Cortex A microarchitectures; in most cases these instructions speed up 32-bit operations, and in some cases they speed up 64-bit operations. (One should not think that *all* CPUs have vector ALUs: a notable counterexample is the Tegra 2, a stripped-down Cortex A9 without ARM's "NEON" vector instructions.) The newest Intel CPUs have 256-bit vector instructions; it is not yet clear whether JH can take advantage of these instructions, and it is reasonably clear that most other SHA-3 candidates cannot.
– Many microarchitectures include instructions that carry out two different arithmetic operations in a single ALU cycle. For example, the x86 and amd64 architectures include an "LEA" instruction that computes $a + sb + c$ (modulo 2^{32} or 2^{64} respectively) where c and s are constants with $s \in \{1, 2, 4, 8\}$; many x86 and amd64 microarchitectures include an ALU that handles this instruction in a single cycle.
– Sometimes an operation can be offloaded to another CPU unit. For example, a typical CPU has a load/store unit that operates in parallel with the ALUs; often this load/store unit is also able to permute data; often one can use a permutation instruction handled by the load/store unit instead of, e.g., a rotate-64-bits-by-8-bits instruction handled by an ALU.

There are also several overheads that make computations slower:

– ALUs operate only on data in a limited number of "registers". Data must be loaded (copied from RAM into registers) before arithmetic, and stored (copied from registers into RAM) afterwards. A computation that does not fit into registers is often bottlenecked by loads and stores, leaving the ALUs idle.
– ALUs are typically asymmetric: for example, it is rare for a core to have more than one ALU that understands multiplication. A computation involving many operations supported by a single ALU will be bottlenecked by that ALU, leaving the other ALUs idle.
– Sometimes a single instruction requires two or more trips through the same circuitry, occupying the ALU for two or more cycles. For example, Westmere needs two cycles for an AES-round instruction, while Sandy Bridge needs only one cycle; apparently Westmere has 8 parallel single-cycle S-box circuits while Sandy Bridge has 16.
– Sometimes a single operation requires several instructions. For example, the amd64 instruction set includes c-bit rotation of 32 bits, and c-bit rotation of 64 bits, but it does not include vectorized rotations: a 4×32-bit vector rotation uses three instructions (shift, shift, xor) plus a copy (since shift overwrites its input).
– Each instruction has some latency: a number of cycles between the inputs being ready and the output being ready. The CPU core will pause if it cannot find any instructions whose inputs are ready.
– Some microarchitectures are in-order: an instruction cannot start before the preceding instruction has started. All microarchitectures have at least limited forms of the same

rule, even if they are labelled "out-of-order": for example, there is always a limit on the number of instructions that can be reordered at once, and it is common for a load/store instruction to wait for preceding load/store instructions.

3 `blake256`

The main bottleneck in BLAKE-256 is a series of 28 half-rounds for each 64-byte block. Each half-round contains 4 parallel "G functions" each consuming 6 32-bit additions, 4 32-bit rotations, and 6 32-bit xors. Overall this bottleneck contains 10.5 32-bit additions per byte, 7 32-bit rotations per byte, and 10.5 32-bit xors per byte.

On a v6, the 21 additions and xors per byte consume at least 21 cycles per byte. This lower bound ignores the 7 rotations per byte because ARM's addition and xor instructions include free rotations. Schwabe, Yang, and Yang have measured performance of 33.93 cycles per byte; the gap is mainly from load/store overhead.

On a Tegra 2, the same 21 instructions per byte consume at least 10.5 cycles per byte. The measured performance of `blake256` is 30.10 cycles per byte. Presumably the gap is caused by fixable latency problems.

On a typical Intel or AMD CPU, the addition and xor instructions do not include free rotations, and 28 32-bit instructions per byte consume at least 9.333 cycles per byte. However, most implementation work has been focused on another approach that seems to allow higher performance. The 4 parallel G functions are naturally vectorizable, applying the same arithmetic operations in the same order (although some of their inputs are permuted in complicated ways). On typical 128-bit vector units one expects 4 32-bit additions to consume 1 vector instruction, 4 32-bit xors to consume 1 vector instruction, and 4 32-bit rotations to consume 4 vector instructions, so this bottleneck contains $10.5/4 + 7 + 10.5/4 = 12.25$ vector instructions per byte. Half of the rotations are by 8 or 16 bits and can be carried out by permutations, leaving only $10.5/4 + 7/2 + 10.5/4 = 8.75$ vector instructions per byte.

A typical Intel or AMD CPU core can carry out at least 2 vector instructions per cycle, reducing this bottleneck below 5 cycles per byte. However, the measured performance of BLAKE-256 is not nearly this fast:

- blake256, 8.86 cycles/byte: amd64; Westmere (206c2); 2010 Intel Xeon E5620; 4 x 2400MHz; hydra2, supercop-20111120
- blake256, 12.92 cycles/byte: amd64; C2 65nm (6fb); 2007 Intel Core 2 Quad Q6600; 4 x 2405MHz; utrecht, supercop-20111120
- blake256, 13.36 cycles/byte: amd64; K10 32nm (300f10); 2011 AMD A8-3850; 4 x 2900MHz; hydra5, supercop-20111120
- blake256, 14.28 cycles/byte: amd64; K10 45nm (100fa0); 2010 AMD Phenom II X6 1090T; 6 x 3200MHz; hydra1, supercop-20111120

Part of the gap is explained by latency. Each G function modifies 4 32-bit words a, b, c, d as follows, using four auxiliary inputs m_i, k_i, m_j, k_j, as shown in Figure 3.1. The latency from each b through the next b is the total latency of 4 addition-xor-rotate chains; the latency of a series of 28 G computations is the total latency of 112 addition-xor-rotate chains, i.e., 1.75 addition-xor-rotate chains per byte. Even on a CPU offering single-cycle latency for vector instructions, half of the rotations have latency 3 (copy, shift, xor), so the chains have

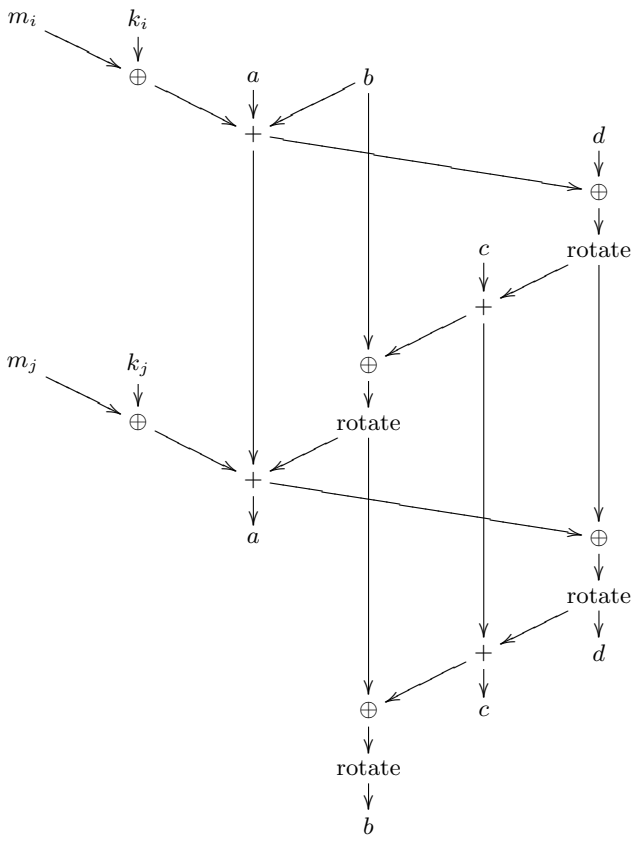

Fig. 3.1. Data flow in BLAKE's G function.

latency 4 on average, for a lower bound of 7 cycles per byte; and many CPUs have more than single-cycle latency.

Note that one can improve rotation latency at the expense of an additional operation. There is ample time to make a copy of d, for example; an additional xor then produces two separate copies of the first rotation input, reducing the rotation latency to 2 (shift, xor) and thus reducing the lower bound to 6.125 cycles per byte. Existing implementations of `blake256` do not seem to have been optimized for latency, and the performance of latency-sensitive implementations is still unclear.

A more straightforward way to work around latency bottlenecks is to hash two blocks in parallel on the same core. One can accomplish this by hashing two separate messages (in applications that have multiple messages to hash at once); by hashing two separate blocks of the same message (using a tree mode or another parallelizable hashing mode); or by hyperthreading. None of these approaches have been benchmarked yet.

4 `blake512`

The main bottleneck in BLAKE-512 is a series of 32 half-rounds for each 128-byte block. Each half-round contains 4 parallel "G functions" each consuming 6 64-bit additions, 4 64-bit rotations, and 6 64-bit xors. Overall this bottleneck contains 6 64-bit additions per byte, 4 64-bit rotations per byte, and 6 64-bit xors per byte.

On a high-power 64-bit CPU from Intel or AMD one would expect BLAKE-512's 16 64-bit operations per byte to consume 5.333 cycles per byte. The measured performance of BLAKE-512 is somewhat worse than this:

- blake512, 7.38 cycles/byte: amd64; Westmere (206c2); 2010 Intel Xeon E5620; 4 x 2400MHz; hydra2, supercop-20111120
- blake512, 7.77 cycles/byte: amd64; K10 32nm (300f10); 2011 AMD A8-3850; 4 x 2900MHz; hydra5, supercop-20111120
- blake512, 7.81 cycles/byte: amd64; C2 65nm (6fb); 2007 Intel Core 2 Quad Q6600; 4 x 2405MHz; utrecht, supercop-20111120
- blake512, 8.37 cycles/byte: amd64; K10 45nm (100fa0); 2010 AMD Phenom II X6 1090T; 6 x 3200MHz; hydra1, supercop-20111120

I have not yet analyzed the gaps here.

5 groestl256

The main bottleneck in Grøstl-256 is a series of 20 rounds for each 64-byte block. These 20 rounds are actually 10 rounds applied to one 512-bit input and 10 rounds separately applied to another 512-bit input. Each round contains 64 parallel "S-boxes" mapping a byte to a byte (as in AES), typically implemented as table lookups. Each round also contains various linear operations that are easily absorbed into the table lookups, replacing the 8-bit table outputs with 64-bit table outputs, although an average table lookup then requires nearly three instructions (shift, mask, load). Overall this bottleneck contains 20 table lookups per byte.

Current high-power Intel CPUs have an AES instruction performing many of these S-box lookups per cycle, but Grøstl is then bottlenecked by the other operations:

- groestl256, 11.29 cycles/byte: amd64; Westmere (206c2); 2010 Intel Xeon E5620; 4 x 2400MHz; hydra2, supercop-20111120
- groestl256, 11.51 cycles/byte: amd64; Sandy Bridge (206a7); 2011 Intel Core i7-2600K; 4 x 3400MHz; threads; sandy0, supercop-20110708

I have not yet analyzed how much room there is for improvement here.

On older Intel CPUs, each table lookup consumes one load/store cycle, so this bottleneck consumes at least 20 cycles per byte. The measured Grøstl performance is close to this:

- groestl256, 22.39 cycles/byte: amd64; C2 65nm (6fb); 2007 Intel Core 2 Quad Q6600; 4 x 2405MHz; utrecht, supercop-20111120
- groestl256, 22.48 cycles/byte: amd64; C2 45nm (10676); 2007 Intel Xeon X5450; 8 x 2992MHz; gcc14, supercop-20111120
- groestl256, 23.83 cycles/byte: amd64; Nehalem (106a5); 2009 Intel Xeon E5504; 8 x 2000MHz; dragon, supercop-20111120

Another implementation strategy, using bitsliced vector instructions, is faster than table lookups for AES on C2 and Nehalem and might also be faster than table lookups for Grøstl.

High-power AMD CPUs have two load/store units but are still bottlenecked by the total number of instructions, and thus cannot run much below 20 cycles/byte. The measured Grøstl performance is very close to this:

- groestl256, 19.43 cycles/byte: amd64; K10 45nm (100fa0); 2010 AMD Phenom II X6 1100T; 6 x 3300MHz; hydra3, supercop-20111120
- groestl256, 19.85 cycles/byte: amd64; K10 65nm (100f23); 2008 AMD Phenom 9550; 4 x 2200MHz; ranger, supercop-20111120

For the same reason, Bobcat and Atom CPUs cannot run much below 30 cycles/byte:

- groestl256, 29.57 cycles/byte: amd64; Bobcat (500f20); 2011 AMD E-450; 2 x 1650MHz; h4e450, supercop-20111120
- groestl256, 69.13 cycles/byte: amd64; Atom (106ca); 2010 Intel Atom N455; 1 x 1000MHz; h2atom, supercop-20111120

Presumably Grøstl triggers larger latency problems on Atom than on Bobcat.

6 `groestl512`

The main bottleneck in Grøstl-512 is a series of 28 rounds for each 128-byte block. Each round contains 128 parallel S-boxes, and various linear operations that are easily absorbed into table lookups. Overall this bottleneck contains 28 table lookups per byte.

The speed ratio between Grøstl-512 and Grøstl-256 is often larger than $28/20 = 1.4$. The main difficulty for Grøstl-512 is that the rounds are working on a 1024-bit state that, with most implementation strategies, does not fit into registers. Many other SHA-3 candidates also have states of 1024 bits or larger, but most of the SHA-3 candidates are bottlenecked by arithmetic rather than by loads.

7 `jh256` and `jh512`

`jh256` and `jh512`, the JH proposals for SHA-3-256 and SHA-3-512, run at the same speed, and are considered together here.

The main bottleneck in JH is a series of 42 rounds for each 64-byte block. Each round begins by dividing the 1024-bit JH state into 256 clumps of 4 bits, and applying a 23-bit-operation S-box to each clump. Each round continues by dividing the state into 128 clumps of 8 bits, and applying a 10-bit-operation linear transformation to each clump. A round thus contains $256 \cdot 23 + 128 \cdot 10 = 7168$ bit operations, i.e., 112 bit operations per byte. Overall this bottleneck contains $112 \cdot 42 = 4704$ bit operations per byte.

Exactly the same operations are applied to each clump, so all of these bit operations are naturally 128-bit vectorizable: this bottleneck contains $4704/128.0 = 36.75$ 128-bit vector operations per byte. The division into clumps varies from round to round, creating some bit-permutation overhead; but most of these permutations are free byte permutations, and the remaining permutations are efficiently handled with a few vector shifts and masks.

Most vector ALUs support negated-AND as a single operation. The 23 bit operations in the S-box include 4 negations; 3 of them are easily integrated into negated-AND, and the 4th can sometimes be eliminated (2011 Schwabe–Yang–Yang), leaving 19 bit operations and thus 31.5 128-bit vector operations per byte. This bottleneck consumes at least

- 10.5 cycles per byte on Sandy Bridge, Westmere, Nehalem, and C2, which have three 128-bit vector ALUs;

- 15.75 cycles per byte on K10 and Atom, which have two 128-bit vector ALUs;
- 21 cycles per byte on K8, which has three 64-bit ALUs;
- 31.5 cycles per byte on Cortex A and Bobcat, which have one 128-bit vector ALU;
- 63 cycles per byte on Tegra 2, which has two 32-bit ALUs; and
- 126 cycles per byte on v6, which has one 32-bit ALU.

Most of the round-3 JH measurements are from a slow third-party implementation. This implementation uses only 32-bit operations and does not make serious efforts to reduce load/store overhead. It cannot be expected to beat 42 cycles per byte (on CPUs with three ALUs), and actually runs at around 47 cycles per byte.

One better-optimized implementation has been reported for v6, taking about 156 cycles per byte. The gap between 126 and 156 is explained mainly by load/store overhead; v6 cannot overlap load/store with arithmetic.

For round-2 JH the analogous bottleneck consumes 9 cycles per byte on Sandy Bridge and 13.5 cycles per byte on K10. The round-2 JH implementations from the JH designer are around 18 cycles per byte.

8 `keccakc512` and `keccakc1024`

`keccakc512` and `keccakc1024`, the Keccak proposals for SHA-3-256 and SHA-3-512, run in the same time per block but have different block sizes, 136 bytes and 72 bytes respectively. The following description considers bottlenecks in `keccakc512`; the same bottlenecks appear in `keccakc1024`, magnified by a factor 136/72.

The main bottleneck in Keccak is a series of 24 rounds for each 136-byte block. Each round transforms 25 64-bit words in place, using 76 xors, 29 rotations, and 25 negated ANDs. Overall this bottleneck contains 13.941 64-bit xors per byte, 5.118 64-bit rotations per byte, and 4.412 64-bit negated ANDs per byte.

On a high-power 64-bit CPU from Intel or AMD one would expect Keccak's 23.471 64-bit operations per byte to consume 7.824 cycles per byte. Keccak's measured performance is somewhat worse than this:

- keccakc512, 11.70 cycles/byte: amd64; C2 65nm (6fb); 2007 Intel Core 2 Quad Q6600; 4 x 2405MHz; utrecht, supercop-20111120
- keccakc512, 12.10 cycles/byte: amd64; K10 32nm (300f10); 2011 AMD A8-3850; 4 x 2900MHz; hydra5, supercop-20111120
- keccakc512, 12.25 cycles/byte: amd64; Westmere (206c2); 2010 Intel Xeon E5620; 4 x 2400MHz; hydra2, supercop-20111120
- keccakc512, 12.25 cycles/byte: amd64; K10 45nm (100fa0); 2010 AMD Phenom II X6 1100T; 6 x 3300MHz; hydra3, supercop-20111120

The main problem appears to be load/store overhead, but I have not yet performed a detailed analysis.

9 `skein512256` and `skein512512`

`skein512256` and `skein512512`, the Skein proposals for SHA-3-256 and SHA-3-512, run at the same speed.

The main bottleneck in Skein is a series of 72 rounds for each 64-byte block. Each round is advertised as containing 4 parallel "MIXes" each consuming a 64-bit addition, a 64-bit rotation, and a 64-bit xor; but each round also has 4 key additions, each consuming another 64-bit addition. Overall this bottleneck contains 9 64-bit additions per byte, 4.5 64-bit rotations per byte, and 4.5 64-bit xors per byte.

On a high-power 64-bit CPU from Intel or AMD one would expect Skein's 18 64-bit operations per byte to consume 6 cycles per byte. Skein's measured performance is close to this. For example:

- skein512512, 6.06 cycles/byte: amd64; K10 32nm (300f10); 2011 AMD A8-3850; 4 x 2900MHz; hydra5, supercop-20111120
- skein512512, 6.12 cycles/byte: amd64; K10 45nm (100fa0); 2010 AMD Phenom II X6 1100T; 6 x 3300MHz; hydra3, supercop-20111120
- skein512512, 6.20 cycles/byte: amd64; Westmere (206c2); 2010 Intel Xeon E5620; 4 x 2400MHz; hydra2, supercop-20111120
- skein512512, 6.40 cycles/byte: amd64; C2 65nm (6fb); 2007 Intel Core 2 Quad Q6600; 4 x 2405MHz; utrecht, supercop-20111120

On a low-power 64-bit Bobcat or 64-bit Atom one would expect Skein's 18 64-bit operations per byte to consume 9 cycles per byte. Skein's measured performance is closer to this on Bobcat than on Atom:

- skein512512, 9.10 cycles/byte: amd64; Bobcat (500f20); 2011 AMD E-450; 2 x 1650MHz; h4e450, supercop-20111120
- skein512512, 10.01 cycles/byte: amd64; Atom (106ca); 2010 Intel Atom N455; 1 x 1000MHz; h2atom, supercop-20111120
- skein512512, 12.14 cycles/byte: amd64; Atom (106ca); 2009 Intel Atom D510; 2 x 1667MHz; threads; gcc47, supercop-20111120

Presumably Skein triggers larger latency problems on Atom than on Bobcat.

On a v6, each 64-bit addition uses two 32-bit instructions (add, add-with-carry), and each 64-bit rotate-and-xor uses four 32-bit instructions. Skein's 18 64-bit operations per byte thus consume at least 36 32-bit instructions per byte. Schwabe, Yang, and Yang have measured performance of 42.10 cycles per byte; the gap is mainly from load/store overhead.

On a Tegra 2, the same 36 32-bit instructions per byte consume at least 18 cycles per byte. Skein's measured performance is 37.62 cycles per byte. Presumably the gap is caused by fixable latency problems.

On a Cortex A, a different implementation strategy is better. The 128-bit vector instructions include 1-cycle 2×64-bit additions, 1-cycle 2×64-bit logical operations, and 2-cycle 2×64-bit shifts. In effect each 64-bit addition takes 0.5 cycles, and each 64-bit rotate-and-xor takes 3 cycles (two shifts, two xors), so this bottleneck consumes $9 \cdot 0.5 + 4.5 \cdot 3 = 18$ cycles per byte. This might sound equivalent to the 32-bit approach, but some shifts are free vector permutations, reducing the lower bound below 18 cycles per byte. Skein's measured performance is 15.23 cycles per byte.

References

[1] Daniel J. Bernstein, Tanja Lange (editors), *eBASH: ECRYPT Benchmarking of All Submitted Hashes (accessed 3 January 2012)*, 2012. URL: http://bench.cr.yp.to. Citations in this document: §1.