SHA-256 in FPGA

Damian B. Fedoryka

***Abstract*— SHA-256, or Secure Hash Algorithm-256, is one of the latest hash functions standardized by the U.S. Federal Government. This paper outlines an implementation of this new standard in FPGA. First, the standard is defined, followed by a description of our design and implementation. Finally the results of the synthesis tests are given along with a comparison to similar tests on a comparable FPGA implementation of the previous FIPS standard, SHA-1, as well as another new standard, SHA-512.**

***Index Terms*—collision resistance, digest, hash, signature**

I. Introduction

Hash functions map a message of arbitrary length to n-bit *hash values*, called a *message digest.* A hash function is typically a serially looped function in which subsequent intermediate blocks are dependent on previous blocks. As a result any change in the message will result in a change in the digest with high probability. Hash functions are used primarily in message authentication, where the hash of a given message acts as a "fingerprint" for that message. A cryptographically strong hash function has the following properties[1]:

One-way property – given a code h, it is computationally infeasible to find an x such that $H(x) = h$.

Weak collision resistance – given a block x, it is computationally infeasible to find y such that $H(x) = H(y)$.

Strong collision resistance –computationally infeasible to find a pair (x,y) , $x \neq y$, such that $H(x) = H(y)$.

An attack against finding a message corresponding to a given n-bit hash (property a) requires computations on the order of about 2^n. On the other hand, according to the birthday paradox[2], a collision attack requires significantly fewer computations – $2^{n/2}$ attempts result in a 50% probability of success. As a result an n-bit hash function provides a level of security of n/2 bit against collision attacks. Until recently, the Federally standardized hash algorithm was SHA-1, a 160-bit hash offering 80 bits of collision resistance. With the recent introduction of AES, offering security levels of 128, 192 and 256-bits, SHA-1 no longer offers a level of security to match the encryption standard. Three new standards were approved to match the three security levels of AES: SHA-256, SHA-384 and SHA-512.

[1] There is no set naming convention for these properties. Often two or three properties are combined under a single title.

[2] The birthday paradox states that the probability of any two persons in a group who share any birthday is significantly greater than finding a person in that group whose birthday matches a particular member of the group. In the latter case, the probability surpasses 0.5 in a group of only 28 persons.

II. SHA-256 Description

A. General

SHA-256 accepts messages with arbitrary lengths up to 2^{64}-bits. The message is divided into uniform-size blocks each of which is run through a compression function loop of 64 iterations. Intermediate hash values are rerouted back into the compression loop. The final hash output is 256-bits in length.

B. Preprocessing

As with other popular hashing functions, with SHA-256 the message to be hashed is first padded so that its final length is a multiple of 512 bits. The message is padded as follows: given a message M of length l, append a "1" to the message followed by k zeros, where k is the smallest non-negative solution to the equation $l + 1 + k \equiv 448 \bmod 512$, followed by a 64-bit block whose value is l in binary form. The message is then parsed into N 512-bit blocks, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. Each block i is divided into 15 sub-blocks, $M^{(i)}_0, M^{(i)}_1, \ldots M^{(i)}_{15}$.

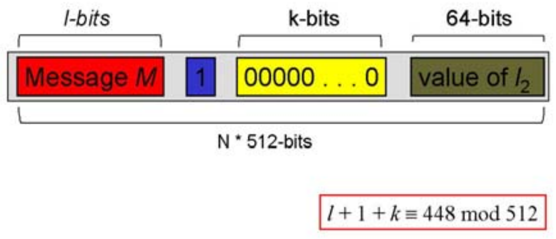


Fig. 1a Message preprocessing: parsing.

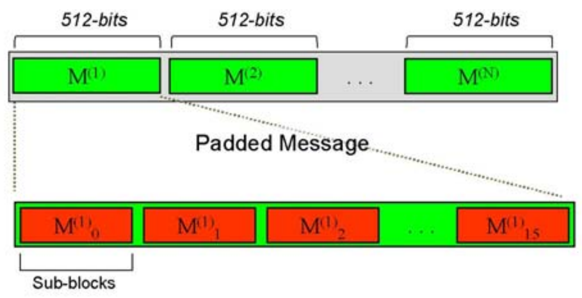


Fig. 1b Message preprocessing: padding

C. *Definitions*

SHA-256 uses the following six logical functions:

$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$
$Maj\,(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\Sigma_0(x) = S^{2}(x) \oplus S^{13}(x) \oplus S^{22}(x)$
$\Sigma_1(x) = S^{6}(x) \oplus S^{11}(x) \oplus S^{25}(x)$
$\sigma_0(x) = S^{7}(x) \oplus S^{18}(x) \oplus R^{3}(x)$
$\sigma_1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)$

where

$\wedge$ = bit-wise OR
$\oplus$ = bit-wise XOR
$\neg$ = bit-wise inversion
S^{i} = *i*-bit right rotate
R^{i} = *i*-bit right shift

D. *The Algorithm*

The message, M is expanded by a *Message Scheduler* according to the following function:

```
For $j$ = 0,1,2, . . . , 15: W = $M_j^{(i)}$ and
For $j$ = 16 to 63
{
   $W_j \leftarrow \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$
}
```

SHA-256 uses a set of 64 defined constants[3], K_1 K_2, . . ., K_{63} [FIPS].
The hash function proceeds according to the following:

```
For i = 1 to N
{
  Initialize registers a,b,c,d,e,f,g,h with the (i-1)st intermediate hash value
  Apply the following compression function to registers a-h:
  For j = 0 to 63
  {
      $T_1 \leftarrow h + \Sigma_1(e) + Ch(a,b,c) + K_j + W_j$
      $T_2 \leftarrow \Sigma_0(a) + Maj(a,b,c)$
      $h \leftarrow g$
      $g \leftarrow f$
      $f \leftarrow e$
      $e \leftarrow d + T_1$
      $d \leftarrow c$
      $c \leftarrow a$
      $b \leftarrow a$
      $a \leftarrow T_{1\,+}\,T_2$
  }
  ith intermediate hash:
  $H_1^{(i)} \leftarrow a + H_1^{(i-1)}$
  $H_2^{(i)} \leftarrow b + H_2^{(i-1)}$
  …
  $H_8^{(i)} \leftarrow h + H_8^{(i-1)}$
}
```

The hash of M:
$H^{(N)} = (H_1^{(N)}, H_2^{(N)}, . . . , H_8^{(N)})$

[3] The constants are derived from the cubed root of the first 64 primes.

III. Design

A. *Basic Architecture*

The upper-most level of hash function architecture consists of four main units: preprocessing unit, message scheduler, message digest and control unit (cf. Fig. 1).

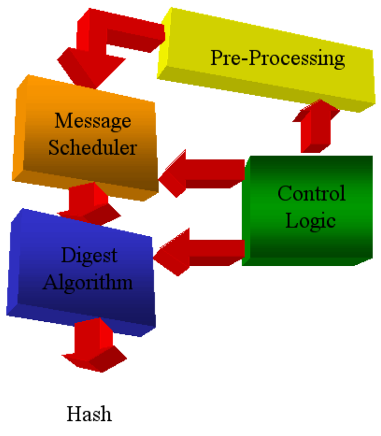


Fig. 2 Block diagram of SHA-256

The message scheduler is depicted in Fig. 2. For cycles 1-16 the mux outputs the unexpanded message. After 16 clocks all 16 registers are loaded and the mux outputs the value of the rotated/shifted and added register values. The mux is reset after 64 clocks. The message scheduler output is fed into the W input of the digest unit. The digest unit (Fig. 3) begins on clock 0 with the V_0 value selected on mux0. These 8 32-bit vectors are loaded into the register bank, which consists of 8 32-bit registers. These registers are loaded once per 512-bit message block (on clock=0), at which point mux0 outputs V_i instead of V_0. Mux1 selects the output of reg0 on clock=0; for clock=1 to 63, mux1 selects the looped back intermediate hash value. Figure 4 illustrates the architecture of the compression function. The function computes once per clock.

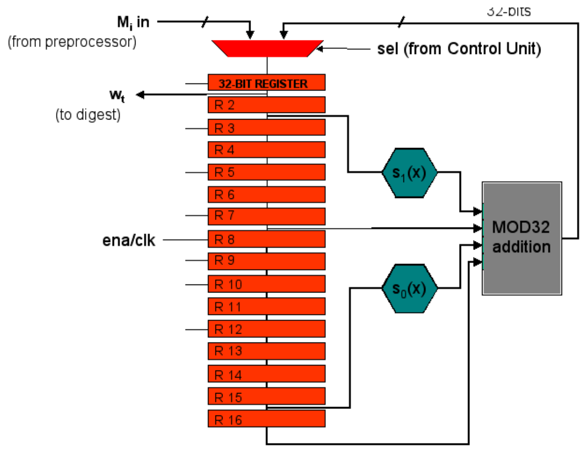
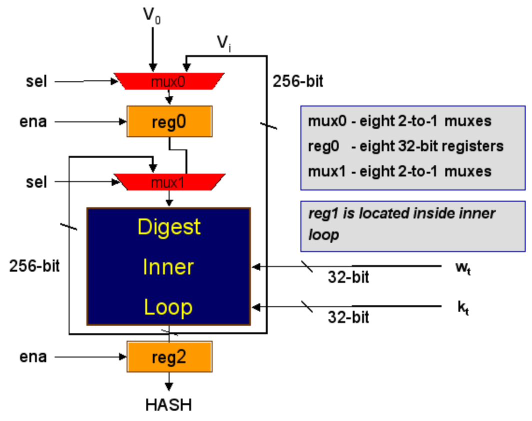


Fig. 3 Message scheduler



V0
Vi
sel
mux0
256-bit
ena
reg0
mux0 - eight 2-to-1 muxes
reg0 - eight 32-bit registers
mux1 - eight 2-to-1 muxes
sel
mux1
reg1 is located inside inner loop
Digest
Inner
Loop
256-bit
32-bit
wt
32-bit
kt
ena
reg2
HASH

Fig. 4 High-level description of Message Digest Unit

On the 64th iteration the current register values must be added to the V_i vector. This addition occurs in the initial vector addition box of Fig. 4.

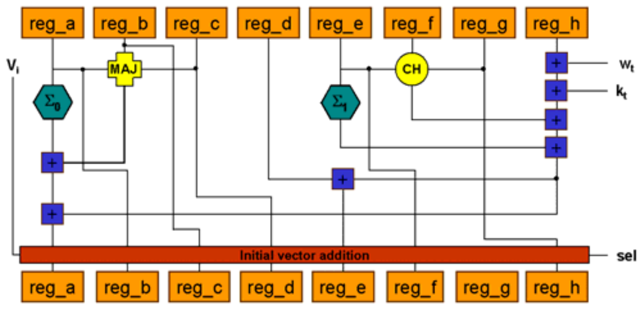


Fig. 5 Inner compression loop (un-optimized)

Figure 5 shows a detail of the V_i addition process. Using mux2, a zero vector (32 zeros) is added to the hash value on each cycle. Only on the 63rd cycle does the mux select the output of register_0 instead of the zero vector. Although it adds an addition to every compression loop, this design prevents the need to add an extra clock cycle after each intermediate hash computation for the addition of the V_i vector.

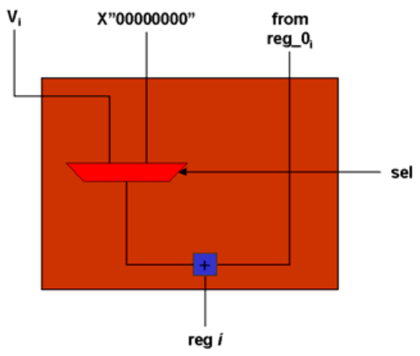


Fig. 6 Initial vector addition module

Figure 6 illustrates the control logic used to set the muxes. These signals are also used to enable the registers.

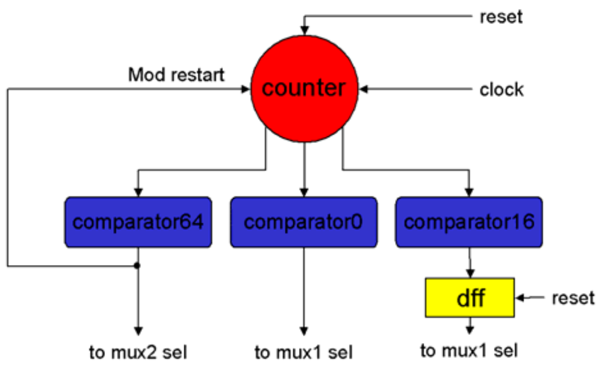


Fig. 7 Control architecture

IV. Simulation and Testing

A. Coding

The unit was modeled in VHDL on Xilinx ModelSim XE Starter. The code for all units was completed with the exception of the preprocessing unit. The high-level entity sha_256 consisted of components corresponding to the block diagram descrption of the algorithm (cf. Fig 2): Message scheduler, control module and digest. The major components of all entities were 32-bit registers, 32-bit 2-to-1 muxes. 32-bit comparators were used in the control module to switch muxes and enable registers. All shift and rotate functions were achieved by simple signal re-routing, either in the structural code or as separate components (as opposed to functions). In the current design the control module utilizes a single counter that is reset after 64 iterations. All units were tied to a common clock and global reset.

Since the preprocessing unit was not completed, message padding and parsing was done in the test bench. The 64 defined constants were also input through the test bench. These constants will most likely be input by the pre-processing unit. Test vectors were obtained from the FIPS specification paper. Two test vectors were used during simulation – one to represent single block messages, the other for multi-block messages. The single-block message was the text "abc" while the multiblock message consisted of the text "abcdbcdecdefdefgefghfghighijhijkijkljklmkl-mnlmnomnopopq." Intermediate and final hash values for these vectors were provide in the FIPS specification for verification.

B. Simulation

The clock period was arbitrarily set at 20 ns. Message and SHA constants were input to the high-level entity on the first clock rise after reset went low. Register_1 was loaded with V_0 on count 1. The intermediate hash value was read from the output of register_2 after a one-clock delay to allow for multi-operand addition delay. On count = 64 the current message block is completed and mux2 is switched to add the Vi vector to the intermediate hash value. This value is loaded into register_0 and loop begins again for next block. The final hash value is available at register_2 after N block iterations.

At the time this paper was being drafted the simulation yielded correct intermediate hash values until clock = 16 at which point the expanded message is input to the digest. After this point intermediate hash values did not match the example values. Debugging indicates either a timing mismatch between the scheduler and digest unit or improper message expansion.

C. Test Results

The target FPGA device chosen was the Xilinx Virtex XCV-1000-C. It is composed of 27,648 Configurable Logic Cells (CLB) slices and has a maximum performance rate of up to 200 Mhz. Timing and resourced analysis was performed on Synplicity's Synplify 9.0. The results of the timing and resource analysis are as follows:

Per cycle propagation delay:	25.289 ns (worst)
Calculated frequency:	39.5 MHz
Throughput: (Block size * Frequency ÷ Digest Rounds)	316 Mbps
Total mapping summary[4]:	1038 LUTs

Table 1 shows a comparison to similar implementations[5] of SHA-1 and SHA-512.

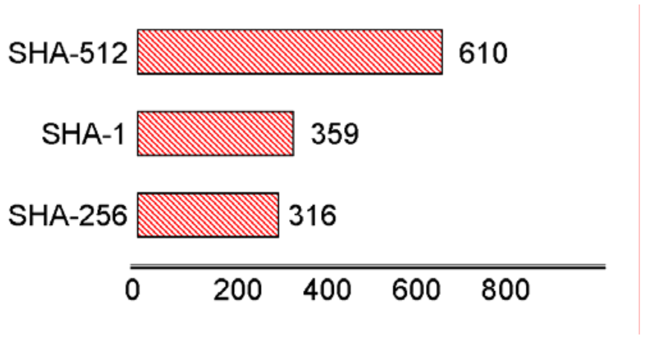


Fig. 8 Throughput comparison to other implementations [Mbps]

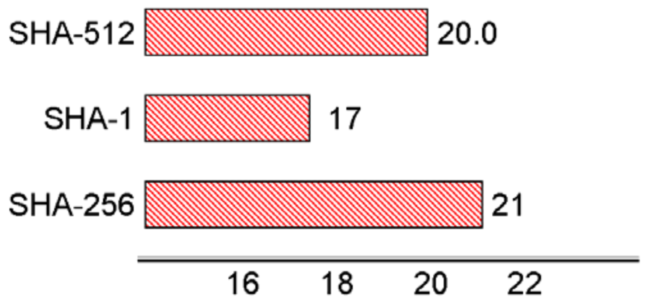


Fig. 9 Minimum clock period comparison to other implementations [ns]

D. Adder Operand Optimization

In a basic architecture, the critical path of the circuit begins at reg_h, travels through the four CPAs and through the CPA on the *a*-bus/*d*-bus for a total of 32 bits x 5 FA delays = 160 FA delays. By using a 5-to-3 and 3-to-2 *CSA counter* design (Fig. 7) the critical path can be reduced to 3 FA + 32 FA CSA counters implemented.

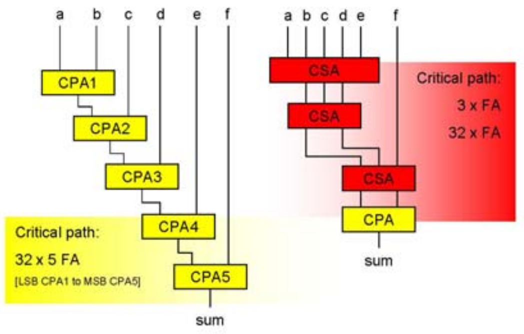


Fig. 10 Six operand addition with and without CSA reduction

[4] Synplify results yielded a total usage of 944 LUTs/61%. Since the preprocessor unit was not implemented at testing time, an additional estimated 10% was added to allow for it.

[5] Note: Both the SHA-1 and SHA-512 implementations utilized a 5-to-3 counter optimization.

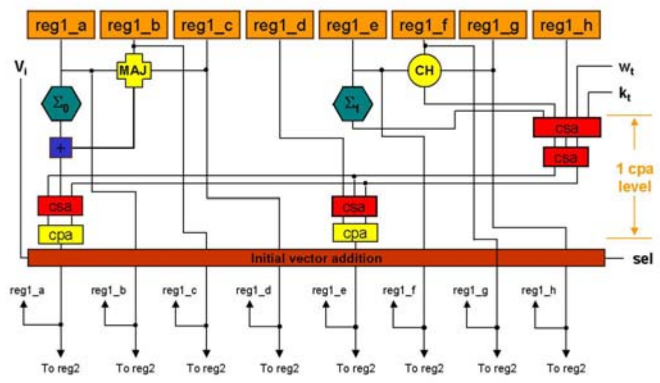


Fig. 11 Digest with 5-to-3 & 3-to-2 Counters

This optimization design was not functioning proprerly as this paper was being drafted. Based on the reduced carry propagation delays, a conservative estimate in clock period reduction is 20% (4 levels instead of 5), or 20.23 ns. This would yield a throughput rate of 395 Mbps (Figs. 8 & 9). Of course these results have not been attained and would have to be verified.

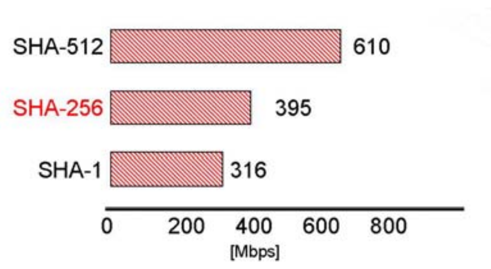


Fig. 12 Digest with 5-to-3 & 3-to-2 Counters

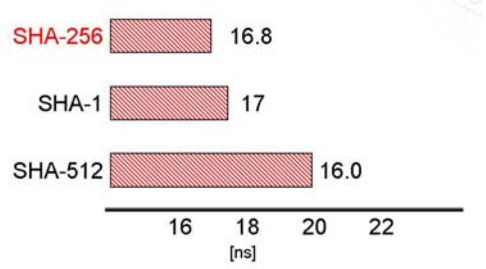


Fig. 13 Digest with 5-to-3 & 3-to-2 Counters

E. Future Development

This implementation of the hash algorithm is an implementation of a basic architecture. There are a number of optimizations that should result in significant improvements, including an increase in bandwidth and a decrease in circuit area. Future development plans for this implementation include the use of a 5-to-3 Counter in the digest unit. The Counter will reduce the critical path in the digest unit from five CPAs to three. Bit-slices of the counter can be implemented on a single CLB slice of the XCV-1000-C, so circuit areas will not increase significantly with this optimization. An additional optimization is unrolling the architecture of the algorithm, which should yield significantly higher throughput rates. Since circuit area on this initial implementation most likely contains redundancy, it is expected that some reduction of area will be achieved through further development. Future plans also include experimental testing using the PCI FPGA board, SLAAC-1V.

V. Summary

An initial basic and un-optimized implementation of this newly standardized hash algorithm demonstrates that the higher security offered by SHA-256 does not necessarily come at a cost to speed and area. Certainly future optimization will improve the results of testing. Future design optimizations and testing should show that the SHA-256 algorithm offers some advantage over the more secure SHA-384 and SHA-512.

Acknowledgment

A special thanks to Dr. Kris Gaj for his support and guidance. Thanks also to Msrs. Roar Lien, Tappan Desai and Pawel R. Chodowiec for their assistance in the labs.

References

[1] W. Stallings, "Cryptography and Network Security," 2nd Edition, Upper Saddle River, NJ: Prentice-Hall, Inc., 1999.

[2] P.J. Ashendon, “The Designer’s Guide to VHDL,” San Francisco, CA: Morgan Kaufman, 1996.

[3] FIPS 180-2 Secure Hash Standard – http://wwcsrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[4] K. Gaj, et al., “Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512,” Fairfax, VA: George Mason University, 2001.