

CE-412 Digital Systems II  
Project Proposal: Secure Hash Algorithm 2 in Verilog

Professor Girma Tewolde, Ph.D.  
Jeremy Maxey-Vesperman  
Zach Butler  
2017-11-10

## **Project Overview**

Zach Butler and Jeremy Maxey-Vesperman would like to create an implementation of the United States National Security Agency's 256-bit Secure Hash Algorithm 2 in Verilog, using the Intel EP2C35F672C6 field programmable gate array as a target. We will implement the pseudocode attached in Appendix 1 using Verilog.

## **Project Description**

The Internet has completely changed the world by connecting everyone. The World Wide Web represents a disruptive technology because, for the first time in history, everyone has an equal ability to disseminate information. Blockchain represents the next socially disruptive technology because, in the same way that the World Wide Web enabled the exchange of information over the Internet, blockchain enables the transfer of value over the Internet. But another facet is even more profoundly disruptive than its applications in the transfer of value. Blockchain also enables decentralized trust for the first time in human history.

Blockchain-based cryptocurrencies have existed for eight years. Users are rewarded for processing others' transactions by being awarded newly minted coin in a process called "mining". In 2013 mining became extremely profitable, so hardware miners were rapidly prototyped on field programmable gate arrays (FPGAs) and sold. Application-specific integrated circuits (ASICs) quickly followed for hardware miners, making FPGA-based mining technology obsolete in year or so.

Nonetheless, implementing a hardware mining solution on an FPGA remains an excellent academic opportunity to study blockchain technology and the algorithms used for real world cryptocurrencies on the Internet. Unfortunately, due to time limitations of the Kettering school term, we have decided to focus on one part of the algorithm. SHA-256 along with RIPEMD-160 make bitcoin and all bitcoin derivative altcoins possible. SHA-256 is also used for most SSL and TLS encryption over HTTPS on the Internet.

## **Required Materials**

- ☐ One Intel EP2C35F672C6 field programmable gate array Cyclone II Altera DE2 development board
- ☐ Altera Quartus II Web Edition, version 11
- ☐ Notepad++
- ☐ Secure Hash Algorithm 2 patent, whitepaper, or pseudocode

## **Expected Outcomes**

We expect to be able to demonstrate a working SHA-256 algorithm on an FPGA using either the board's switches and LEDs, or a serial console to communicate with the FPGA by 11<sup>th</sup> Tuesday, 2017-12-12.

## Appendix 1: SHA-256 Pseudocode

*Note 1: All variables are 32-bit unsigned integers and addition is calculated modulo  $2^{32}$*

*Note 2: For each round, there is one round constant  $k[i]$  and one entry in the message schedule array  $w[i]$ ,  $0 \leq i \leq 63$*

*Note 3: The compression function uses 8 working variables,  $a$  through  $h$*

*Note 4: Big-endian convention is used when expressing the constants in this pseudocode, and when parsing message block data from bytes to words, for example, the first word of the input message "abc" after padding is 0x61626380*

*Initialize hash values:*

*(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):*

$h_0 := 0x6a09e667$

$h_1 := 0xbb67ae85$

$h_2 := 0x3c6ef372$

$h_3 := 0xa54ff53a$

$h_4 := 0x510e527f$

$h_5 := 0x9b05688c$

$h_6 := 0x1f83d9ab$

$h_7 := 0x5be0cd19$

*Initialize array of round constants:*

*(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):*

$k[0..63] :=$

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,  
0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,  
0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,  
0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,  
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,  
0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,  
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,  
0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,  
0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,  
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,  
0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6fff,  
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,  
0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

### *Pre-processing:*

- 1) begin with the original message of length  $L$  bits
- 2) append a single '1' bit
- 3) append  $K$  '0' bits, where  $K$  is the minimum number  $\geq 0$  such that  $L + 1 + K + 64$  is a multiple of 512
- 4) append  $L$  as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits

### *Process the message in successive 512-bit chunks:*

break message into 512-bit chunks

**for** each chunk

create a 64-entry message schedule array  $w[0..63]$  of 32-bit words

*(The initial values in  $w[0..63]$  don't matter, so many implementations zero them here)*

copy chunk into first 16 words  $w[0..15]$  of the message schedule array

*Extend the first 16 words into the remaining 48 words  $w[16..63]$  of the message schedule array:*

**for**  $i$  **from** 16 to 63

$s0 := (w[i-15] \text{ rightrotate } 7) \text{ xor } (w[i-15] \text{ rightrotate } 18) \text{ xor}$   
 $(w[i-15] \text{ rightshift } 3)$

$s1 := (w[i-2] \text{ rightrotate } 17) \text{ xor } (w[i-2] \text{ rightrotate } 19) \text{ xor}$   
 $(w[i-2] \text{ rightshift } 10)$

$w[i] := w[i-16] + s0 + w[i-7] + s1$

*Initialize working variables to current hash value:*

$a := h0$

$b := h1$

$c := h2$

$d := h3$

$e := h4$

$f := h5$

$g := h6$

$h := h7$

*Compression function main loop:*

```
for i from 0 to 63
    S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch := (e and f) xor ((not e) and g)
    temp1 := h + S1 + ch + k[i] + w[i]
    S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj := (a and b) xor (a and c) xor (b and c)
    temp2 := S0 + maj

    h := g
    g := f
    f := e
    e := d + temp1
    d := c
    c := b
    b := a
    a := temp1 + temp2
```

*Add the compressed chunk to the current hash value:*

```
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h
```

*Produce the final hash value (big-endian):*

```
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6
append h7
```

Retrieved 2017-11-09 at 12:38 PM from <https://en.wikipedia.org/wiki/SHA-2#Pseudocode>