**ECE 418 – Network & Computer Security**

**Week 1 Report**

**Implementing SHA-256 on an FPGA in C**

Ally Phelan, John Pitkin, and Addison Wand

Apr 11, 2022

**Abstract**

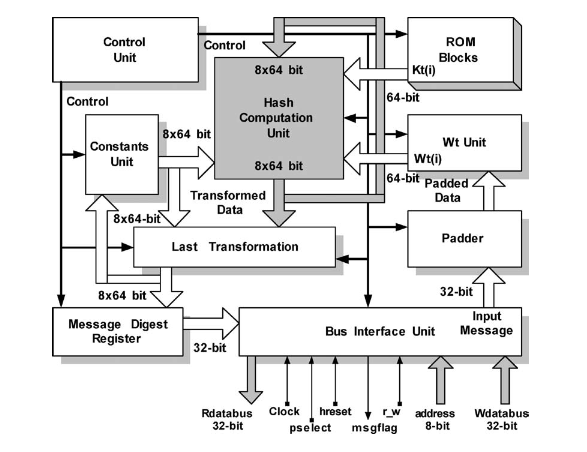
As technology enters more parts of everyday life, cryptography is quickly becoming one of the most important topics in the world. With the internet of things in high demand, more technology; and thus, a larger part of life, is more hackable than ever before. Securing information as it passes through the vastness of the internet has become more and more prudent as hackers and the malware they employ becomes more efficient. The never-ending race between those who want to help and those who want to hurt is raging on. By studying a variety of hashing algorithms, we can preserve the integrity of our messages and authenticate those who send them. This report presents a design for the SHA-256 algorithm implemented on an FPGA, including a high-level description of the design and requirements that will be met.

**Objective**

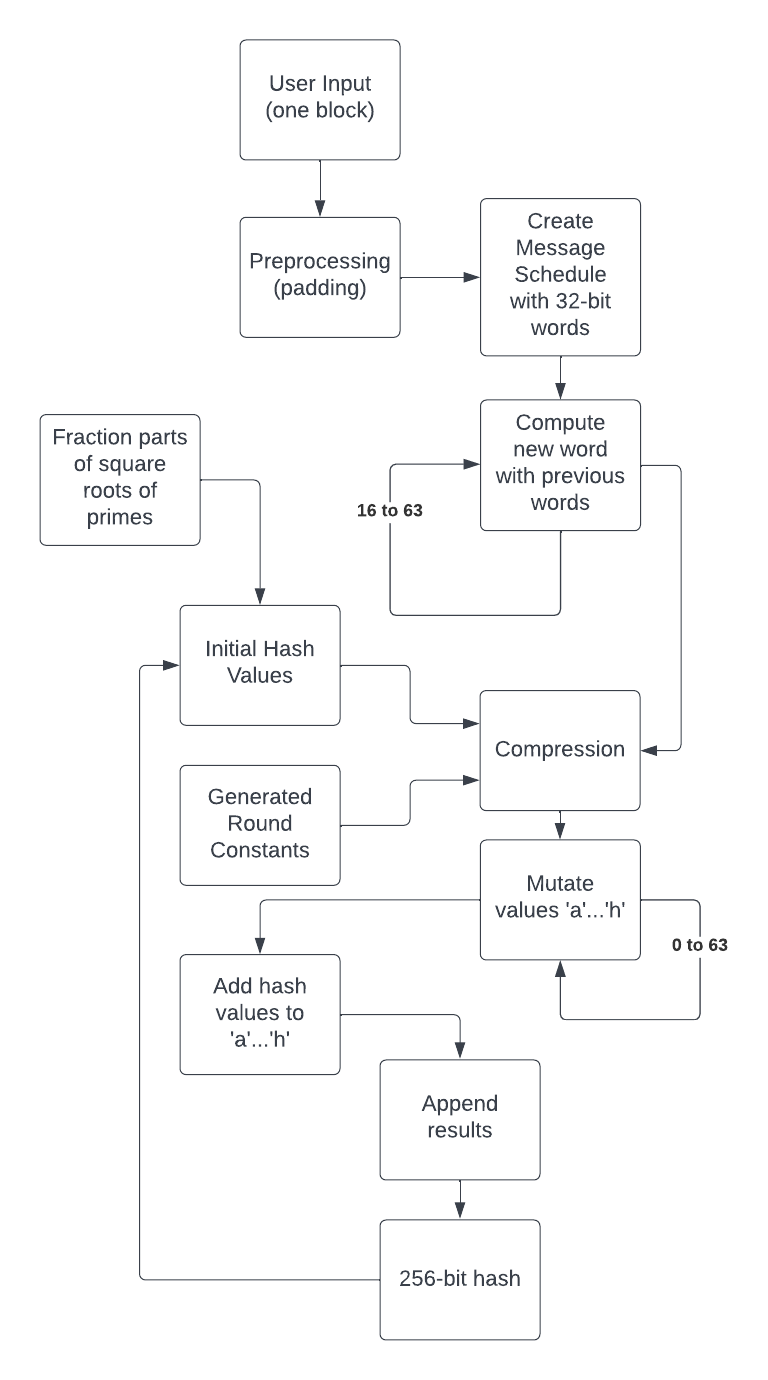
In this project we will be using Sklavos and Koufopavlou’s design for the Implementation of the Sha-2 Hash Family Standard Using FPGAs to guide our implementation of SHA-256 on an FPGA. The FPGA project has the following objectives:

1. Give input via board switches.
2. Transfer input into hash via SHA-256 algorithm.
3. Return hash via LEDs.

In Figure 1 we see the proposed system architecture that the researchers employ in their project.

*Figure 1 -* Proposed System Architecture 

**System Design**

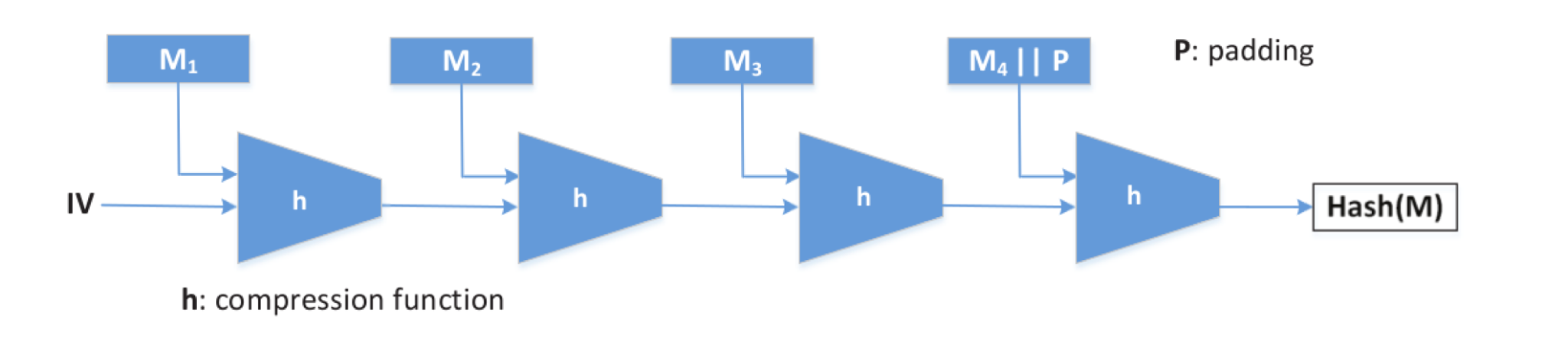


Steps of SHA-256 to be implemented in C

1. Preprocessing - padding out the message with zeros so that it is 512 bits
   1. A single 1 is used to separate the padding and the original message
   2. The last 64 bits of the padding is used to represent the length of the original message
   3. A message longer than 512 bits will use more than one message block
      1. The message will be padded to the next multiple of 512 bits
2. Initialize hash values
   1. 8 initial Hash Values are created from the first 32 bits of the fractional parts of the square roots of the first 8 primes
3. Initialize round constants
   1. 64 round constants are created from the first 32 bits of the fractional parts of the cube roots of the first 64 primes

Hash computation begins…

1. Message schedule for each message block
   1. Split the message block into 32 bit words - (w0 to w15)
   2. Expand our message schedule to 64 words, each 32 bits and all zeros - (w16 to w63)
   3. Modify our new words for the message schedule with the following computations
      1. For i from 16 to 63
         1. s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
         2. s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
         3. w[i] = w[i-16] + s0 + w[i-7] + s1
   4. Expanded message schedule will be fed into the main hash function
2. Compression
   1. First…
      1. We use 8 values initialized with from the first 32 bits of the fractional parts of the square roots of the first 8 primes
         1. Each fractional part is multiplied by 2^32 and converted to binary
      2. These values are assigned to a, b, c, d, e, f, g, h respectively
   2. Second…
      1. Mutate the values in our state registers ‘a’ through ‘h’
      2. Start with the first word from our message schedule, the first initialized round constant, and the current state registers to generate our first temporary word
      3. Get the second temporary word from the majority of a, b, and c summed with a right rotated and xor’ed form of ‘a’
   3. Third… one single compression round…
      1. Move all words in the state registers down…
         1. a->b
         2. b->c
         3. c->d
         4. d->e
         5. e->f
         6. f->g
         7. g->h
      2. ‘a’ becomes the summation of a our two temporary words
      3. Lastly we also add ‘e’ and our first temporary word
      4. In the next round, use the next word from the message schedule and the next round constant (w[i] and k[i])
3. Final steps
   1. Add all of our resulting words and the initial hash values generate from from the first 32 bits of the fractional parts of the square roots of the first 8 primes
   2. Take each 32 bit summation and append them to create our digest



Latter blocks of a given message are hashed with the previous hashes made from previous blocks

**Code - First Two Steps**

#include <stdio.h>

// initializing global block size

int blockSize = 512;

// initializing all global hash values

int h0 = 0x6a09e667;

int h1 = 0xbb67ae85;

int h2 = 0x3c6ef372;

int h3 = 0xa54ff53a;

int h4 = 0x510e527f;

int h5 = 0x9b05688c;

int h6 = 0x1f83d9ab;

int h7 = 0x5be0cd19;

// function for getting the length of a string

int getStringLength(char\* str) {

// initializing string length

int strLength = 0;

// variable for checking each char in string

char temp = str[0];

// iterator variable

int i = 1;

while(temp != '\0') {

temp = str[i];

// incrementing string length

strLength++;

i++;

}

return strLength;

}

// converts a given long to a binary string

char\* longToBinary(unsigned long num, char\* binaryBuffer, int buffer\_size) {

binaryBuffer += (buffer\_size-1); // shifting the buffer pointer

// looping over the buffer and adding binary chars

for (int i = 64; i >= 0; i--) {

// checking to see what the least significant bit is

if ( (num & 1) == 0 ) {

// setting the char value at buffer pointer

\*binaryBuffer = '0';

} else {

// setting the char value at buffer pointer

\*binaryBuffer = '1';

}

binaryBuffer--; // shifting buffer pointer

// right shifting num - erase least significant bit

num >>= 1;

}

// return the buffer with binary string

return binaryBuffer;

}

/\*

\* The padding function that pads a given message until it's a multiple of 512

\* and splits the padded message into message blocks of 512 bits

\*/

void pad(char\* binaryMessage, int sizeBits, char messageBlocks[][blockSize+1], int numBlocks){

//+1 is added to avoid issues involving edge cases before padding is added and after the

//'1' separator is added

int newSizeBits = sizeBits+1;

int i = 0;

// makes newSizeBits the size needed to have exactly 64 remaining spots for the size

while(newSizeBits%512 != 448){

newSizeBits++;

}

// creating a buffer for the message and padding

// including the length of the message in bits

char paddedBuff[newSizeBits+64];

// copies the message into the padded buffer

for(i = 0; i<sizeBits; i++){

paddedBuff[i] = binaryMessage[i];

}

printf("\n\n");

printf("copied message: %s\n", paddedBuff);

printf("\n\n");

// adds a trailing '1' after the message and before the padding of 0's

paddedBuff[sizeBits] = '1';

//pads the buffer up until there are only 64 spots left for the length

for(i = sizeBits + 1; i < newSizeBits; i++){

paddedBuff[i] = '0';

}

// getting the size of the initial message in 64 bits

const int BUFFER\_SIZE = 65;

char messageLengthInBinary[BUFFER\_SIZE];

// string ends with null char

messageLengthInBinary[BUFFER\_SIZE-1] = '\0';

longToBinary(sizeBits, messageLengthInBinary, BUFFER\_SIZE-1);

// appending the length of the initial string in binary

for (i = 0; i < 64; i++) {

paddedBuff[newSizeBits+i] = messageLengthInBinary[i];

}

printf("\n\n");

printf("everything padded: %s\n", paddedBuff);

printf("\n\n");

// splitting the string into blocks of 512 bits and null char

printf("\n\n");

for (int k = 0; k < numBlocks; k++) {

for (int j = 0; j < blockSize; j++) {

messageBlocks[k][j] = paddedBuff[ (k \* blockSize) + j];

}

// appending null char to each block

messageBlocks[k][blockSize] = '\0';

}

printf("\n\n");

}

int main() {

// test input message

char\* message = "110101010100010010100011001101010100110011001100110011001100110011010101010100101010101010010010100100101001010101001010010100010101001010101010101010101000100011011111010110111001010101010101010010101010101001010100101010101111111111111010101010101010110100111100010010100011001101010100110011001100110011001100110011010101010100101010101010010010100100101001010101001010010100010101001010101010101010101000100011011111010110111001010101010101010010101010101001010100101010101111111111110101010101010101101001111000100101000110011010101001100110011001100110011001100110101010101001010101010100100101001001010010101010010100101000101010010101010101010101010001000110111110101101110010101010101010100101010101010010101001010101011111111111100000000000000111111111111111100000000000000111111111111010100000000000000011111111111110000000000000001111111111111000000000000001111111111111100000000000000111111111111111111111111111010101010101010101010101010101000000000000000001111111111111111000000000000000111111111111111110000000000000000111111111111111110000000000000011111111111111100000000000000011111111111111000000000000101010101010101010101010010101010111111111111111000000000000000001111111111111110000000000000011111111111111110010100000000000000000000111111111111111100000000000000001111111111111100000000000000001010101000000000000001111111111111110000000000000001111111111111000000000000010101000000000000000000111111111111111111000000000000000000111111111010101010100101010101010101011111000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010111101001";

// getting the length of the binary string

int messageLength = getStringLength(message);

// finding out how many 512-bit message blocks will be needed

int numBlocksNeeded = 0;

// least length of a message after padding a 1 and the message length

int leastPaddedMessageLength = messageLength + 65;

// last block is 447 bits - will pad to 512 exactly

if (leastPaddedMessageLength % 512 == 0) {

numBlocksNeeded = leastPaddedMessageLength / 512;

} else {

// adding one for integer division

numBlocksNeeded = leastPaddedMessageLength / 512 + 1;

}

// creating the array of strings for the message blocks

char messageBlocks[numBlocksNeeded][blockSize+1];

// padding the message and getting the message blocks

pad(message, messageLength, messageBlocks, numBlocksNeeded);

// printing all of the message blocks

printf("\n\n");

for (int i = 0; i < numBlocksNeeded; i++) {

printf("\n\n");

printf ("block %d: %s\n", i, messageBlocks[i]);

}

printf("\n\n");

return 0;

}

**Timeline**

*Week 1: Design Planning and Writing C code for individual algorithm steps*

*Due Fri. April 15 6PM*

During the week 1, please come up with a project proposal including the details like which

cryptographic algorithm you are planning to develop and the flow chart or the block diagram

explaining the higher-level abstraction of the algorithm. Start developing the C code functions

for the individual steps in the algorithm. Make sure that the functions work without any errors.

Deliverables:

• Include a project outline. Develop a plan to implement the algorithm either in the form

of a flow chart or in the form of a block diagram and explain it clearly.

• Develop a C code for implementing at least the first two steps of the algorithm

successfully.

• Submit a written report with all the above-mentioned details and the C code developed.

The code has to be included in a mono-space Courier New Font

*Week 2: Developing C code for all the steps of the algorithm Due Fri. April 22, 6PM*

During the week 2, develop the C code for implementing all the steps of the selected

cryptographic algorithm. Verify the functionality of these by developing their corresponding test

benches and simulating them.

Deliverables:

• Write down a report explaining the status of the project including all the code developed.

• Also include the simulation reports and the testbenches needed to verify the code into the report.

*Week 3: Complete Cryptographic algorithm Design Due Fri. April 29, 6PM*

At this milestone you will have finished and developed cryptographic algorithm, including the top

level function, testbench to simulate the code and demonstrate the working code to the Professor.

Also, generate the synthesis report of the code.

Deliverables:

• Write down a finished project report explaining the details of the project including all the

code developed. The logic of each function has to be explained clearly.

• Also include the synthesis reports and the testbench needed to verify the code into the

report.

*Week 4: Final Project Deliverables Due Fri. May 06, 6PM*

At this milestone, you should have a slide deck explaining the objectives of the project, including

the flow chart or the block diagram of the projects, steps included in the design, process of

implementation, results and suggestions.

Deliverables:

• A slide deck by the team for presenting the entire work done in the project during the

Finals week.

*Finals Week*

The project has to be orally presented by the team during the scheduled time during the final’s

week.

**Conclusion**

Programming SHA-256 on a FPGA can be beneficial for testing and development of the algorithm. It allows easy programming and debugging to ensure the algorithm works and ensures it is producing the correct results. With an FPGA we hope to recreate the results found in the paper as well as taking advantage of the inputs given to us on the FPGA and printing the results onto the LEDs of the FPGA. Overall we hope to match the results in the implementation of the SHA-265 algorithm.