FPGA-based SoC Design

Design and Implementation of a SHA-1 Hardware Accelerator

Hochschule Darmstadt

FSOC Laboratory #1

Juan Valdivieso (Mtr. No. 1119246)

Mateo Ceballos (Mtr. No. 1112948)

Table of Contents

Table of Figures

1 Part #1: SHA-1 Algorithm

1.1 SHA-1 algorithm in C:

Core SHA-1 function to operate in a single pre-processed 512-bit wide input message:

```c
void sha_1(uint32_t *hash_ptr, const uint32_t *message, const uint32_t *prev_hash) {
    // SHA-1 initialization
    uint32_t h0, h1, h2, h3, h4;
    if (prev_hash == NULL) {
        h0 = 0x67452301;
        h1 = 0xEFCDAB89;
        h2 = 0x98BADCFE;
        h3 = 0x10325476;
        h4 = 0xC3D2E1F0;
    } else {
        h0 = prev_hash[0];
        h1 = prev_hash[1];
        h2 = prev_hash[2];
        h3 = prev_hash[3];
        h4 = prev_hash[4];
    }

    const uint32_t *block = (const uint32_t *)message;

    // Initialize the message schedule
    uint32_t w[80];
    for (int i = 0; i < 16; i++) {
        w[i] = (block[i * 4] << 24) | (block[i * 4 + 1] << 16) |
                (block[i * 4 + 2] << 8) | block[i * 4 + 3];
    }

    for (int i = 16; i < 80; i++) {
        w[i] = SHA1_ROTL((w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16]), 1);
    }

    // Initialize hash value for this chunk
    uint32_t a = h0;
    uint32_t b = h1;
    uint32_t c = h2;
    uint32_t d = h3;
    uint32_t e = h4;

    // Comp function
    for (int i = 0; i < 80; i++) {
        uint32_t f, k;
        if (i < 20) {
            f = SHA1_CH(b, c, d);
            k = SHA1_K1;
        } else if (i < 40) {
            f = SHA1_PARITY(b, c, d);
            k = SHA1_K2;
        } else if (i < 60) {
            f = SHA1_MAJ(b, c, d);
            k = SHA1_K3;
        } else {
            f = SHA1_PARITY(b, c, d);
            k = SHA1_K4;
        }

        uint32_t temp = SHA1_ROTL(a, 5) + f + e + k + w[i];
        e = d;
        d = c;
        c = SHA1_ROTL(b, 30);
        b = a;
        a = temp;
    }

    // Update hash values
    h0 += a;
    h1 += b;
    h2 += c;
    h3 += d;
    h4 += e;

    // Set the final hash value
    hash_ptr[0] = h0;
    hash_ptr[1] = h1;
    hash_ptr[2] = h2;
    hash_ptr[3] = h3;
    hash_ptr[4] = h4;
}
```

Using the following constant and function definitions:

```c
// SHA-1 constants
#define SHA1_K1 0x5A827999
#define SHA1_K2 0x6ED9EBA1
#define SHA1_K3 0x8F1BBCDC
#define SHA1_K4 0xCA62C1D6

// Rotate left function
#define SHA1_ROTL(x, n) (((x) << (n)) | ((x) >> (32 - (n))))

// SHA-1 block size in bytes
#define SHA1_BLOCK_SIZE 64

// Internal function ft() of the compression function
// for 0<=t<=19
#define SHA1_CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
// for 20<=t<=39 && 60<=t<=79
#define SHA1_PARITY(x, y, z) ((x) ^ (y) ^ (z))
// for 40<=t<=59
#define SHA1_MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
```

Figure 1 SHA-1 C core function

The following program implements the correct operation of SHA-1 implementation using a function that implements pre-processing for input messaged of arbitrary length:

```c
int main() {
  char message[] = "FSOC23/24 is fun!";

  // Padded message length calculation (consider that a byte is formed by 8 bits)
  uint64_t padded_messageLen = ((strlen(message) / 55) + 1) * 64;

  // To store the hash values
  uint32_t hash[5] = {0};

  // Message padding
  uint32_t *padded_message = preproces_input(hash, (const uint32_t *)message);

  // SHA function calling
  for (size_t offset = 0; offset < padded_messageLen;
       offset += SHA1_BLOCK_SIZE) {
    const uint32_t *block = padded_message + offset;

    if (offset == 0) {
      sha_1(hash, (const uint32_t *)block, NULL);
    } else if (offset > 0) {
      sha_1(hash, (const uint32_t *)block, hash);
    }
  }

  // Print SHA result hash
  printf("SHA-1 hash: ");
  for (int i = 0; i < 5; i++) {
    printf("%08x", hash[i]);
  }
  printf("\n");

  return 0;
}
```

Figure 2 SHA-1 function implementation program.

The pre-processing function is defined as follows:

```c
uint32_t *preproces_input(uint32_t *hash_ptr, const uint32_t *message) {

  // Breaking the message into 512-bit blocks
  const uint8_t *byte_message = (const uint8_t *)message;

  size_t message_length = strlen((const char *)byte_message);

  // Calculate Padded Length
  size_t padded_length =
      ((message_length + 8) / SHA1_BLOCK_SIZE + 1) * SHA1_BLOCK_SIZE;

  // Allocate memory for Padded Message
  uint32_t *padded_message =
      (uint32_t *)malloc(padded_length * sizeof(uint32_t));
  if (padded_message == NULL) {
    // Handle memory allocation failure
    return NULL;
  }

  // Initialize Padded Message with Zeros
  for (size_t i = 0; i < padded_length; ++i) {
    padded_message[i] = 0;
  }

  // Copy Message Data to Padded Message
  for (size_t i = 0; i < message_length; ++i) {
    padded_message[i] = byte_message[i];
  }
```

```c

    // Append '1' Bit to Padded Message
    padded_message[message_length] = 0x80;

    // Adding the original message length in bits
    uint64_t bit_length = (uint64_t)message_length * 8;
    for (int i = 0; i < 8; i++) {
      padded_message[padded_length - 8 + i] =
          (uint8_t)(bit_length >> (56 - i * 8));
    }

    return padded_message;
  }
```

Figure 3 Pre-process input message function

1.2 Theory Questions

A. What is a mathematical one-way function and what are specific applications?

It is a mathematical function where is easy to compute in one direction, but hard to compute in the opposite direction.

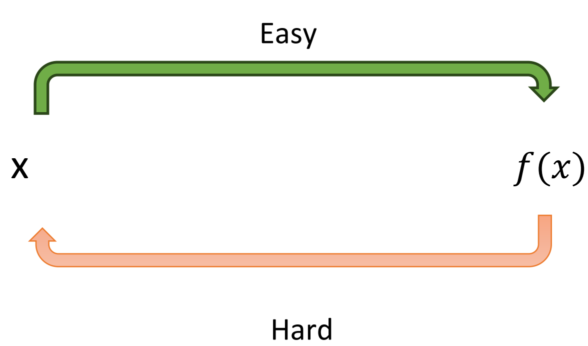


Figure 4 Mathematical One-way function

Given x it is easy to compute f(x). However, given a value y inside the range of f, it is hard to find a x that complies with f(x)=y.

It is said that the equation is hard or impossible to reverse.

Knowing the input, we can get the output of the system. But if we know the output, it is not possible to calculate the input corresponding to that output.

This group of functions are widely used in computer science and cryptography. Typically, hash algorithms take one-way functions to produce output “digest” or “hash” values. The security of cryptographic techniques is based on one-way functions. Some examples of hash applications and one-way function in computer systems are password usage for information access, digital signature protection, blockchain/crypto, data integrity and pseudorandom number generation.

B. Define preimage resistance and the second preimage resistance characteristic of a one-way function.

Preimage resistance is the unfeasibility to find any input (also called preimage) of a mathematical one-way function, that can be mapped to a specific output value.

Considering hash functions, preimage resistance implies that no input can be mapped if the output hash value of a system is known.

The second preimage resistance characteristic of a one-way function specifies that given a value x1 and the corresponding output value f(x1), it is impossible to compute another x2 such that f(x2) = f(x1). This determines that every output is unique to all inputs.

Is the property of a one-way function that states that it is not possible to find any second distinct input that has the same output as a given input.

C. What is a collision and how does it affect the security of a hash function? Why do collisions necessarily exist?

A collision occurs when two different inputs have the same output or hash value. This is a thread for hash security, as data could be accessed, altered or be used by malicious users. Collisions are undesired events that impacts the reliability and efficiency of a hash function.

There are different mathematical principles such as the Pigeonhole principle, that define that collisions are inevitable due to the nature of the hashing process. Theoretically, the existence of a one-way function without any collisions has not been proved. However, hashing functions are desired to be collision free, where finding a collision is computationally infeasible.

D. Does the Boolean XOR function represent a valid way to verify the integrity of a message? Justify your answer!

The Boolean XOR function is not a secure way to verify message integrity. This method is way too simple to ensure 100% integrity, particularly in message security applications. There are different techniques to break the XOR cipher, where the method fails to detect errors during transmission or message manipulations performed by external attackers. Some examples of vulnerability include:

1. Bit-Flipping: If an even number of bits are flipped during transmission, the XOR checksum may remain the same. This error cannot be detected in such bit flipping scenarios.
2. Weak message manipulation detection and lack of avalanche effect: If a small change is performed in the message, XOR integrity verification does not provide a fine differentiation to the location or pattern changes. This mainly happens due to the lack of avalanche effect, where small changes in an input message produce completely different output.
3. Weak collision resistance: finding two different messages that produce the same encrypted data is more feasible.

E. Explain the birthday problem and state the relation to hash collisions.

The birthday problem explains that if we consider a group of people, the probability of finding two people that share the same birthday is higher than we think. First, we must define the following considerations to simplify the problem:

1. No twins are part of the group.
2. No leap years are considered.
3. Every birthday is equally likely.
4. No birthday patterns.

To give some examples of the high probability results:

- If we consider a group of 23 people, there is a 50.73% probability of two people sharing birthdays.
- If we consider a group of 70 people, there is a probability of 99.9% of finding two people that share birthday.

The problem explains that usually, our brains are not very accurate when calculating probabilities that depend on nonlinear functions such as the birthday problem.

The approach to calculate the probabilities of the birthday problem is to first calculate the probability of two people not sharing a birthday. This is a good starting point as we know how many days contains a year. So, the probability of two people not sharing birthday is of 364 days/ 365 days = 0.997 or 99.7%.

To get the probability of two people sharing birthday, we simply subtract:

100% - 99.7% (probability of not sharing birthday)

If we consider a larger group of people, the possible number of combination pairs grows quadratically. That means, the answer is proportional to the square of the number of people.

With the proposed mathematical approach, considering 23 people in a group, the probability of not sharing a birthday can be calculated as follows:

$$\frac{365}{365} * \frac{364}{365} * \frac{363}{365} * \frac{362}{365} * \frac{361}{365} * \frac{360}{365} * \ldots * \frac{343}{365} = 0.492\ (49.2\%)$$

The probability of finding two people that share a birthday is:

$$100\% - 49.2\% = 50.73\%$$

The mathematical approach to this problem helps us to understand that the facts and numbers can show a completely different reality compared to the intuition from our brains.

The birthday problem can be directly related to the collision events in hash functions. Normally, hash functions work with multiple input combinations, where the amount of likelihood to get an equal hash is very high. That is why these types of functions must be proven in detail to avoid such collisions. The problem is even more complicated if we consider that some hash algorithms accept arbitrary length inputs and fixed length output combinations. This explains the complexity and importance of effective hash functions to guarantee security and performance.

F. What role plays the SHA-256 algorithm in the context of the cryptocurrency Bitcoin?

SHA-256 is used for cryptographic security. Is part of the SHA-2 family and a successor of SHA-1. Is considered an extremely secure hash function, because the produced hash values are almost irreversible and unique. It converts input text of any length to 256-bit alphanumeric hash values.

In relationship to cryptocurrency Bitcoin, SHA-256 is a fundamental component of the major blockchain protocols such as Bitcoin and Bitcoin SV. This hash function is used in the proof-of-work validation. This allows the transaction verification within blockchain protocols. SHA-254 assures the security and integrity of the Bitcoin blockchain.

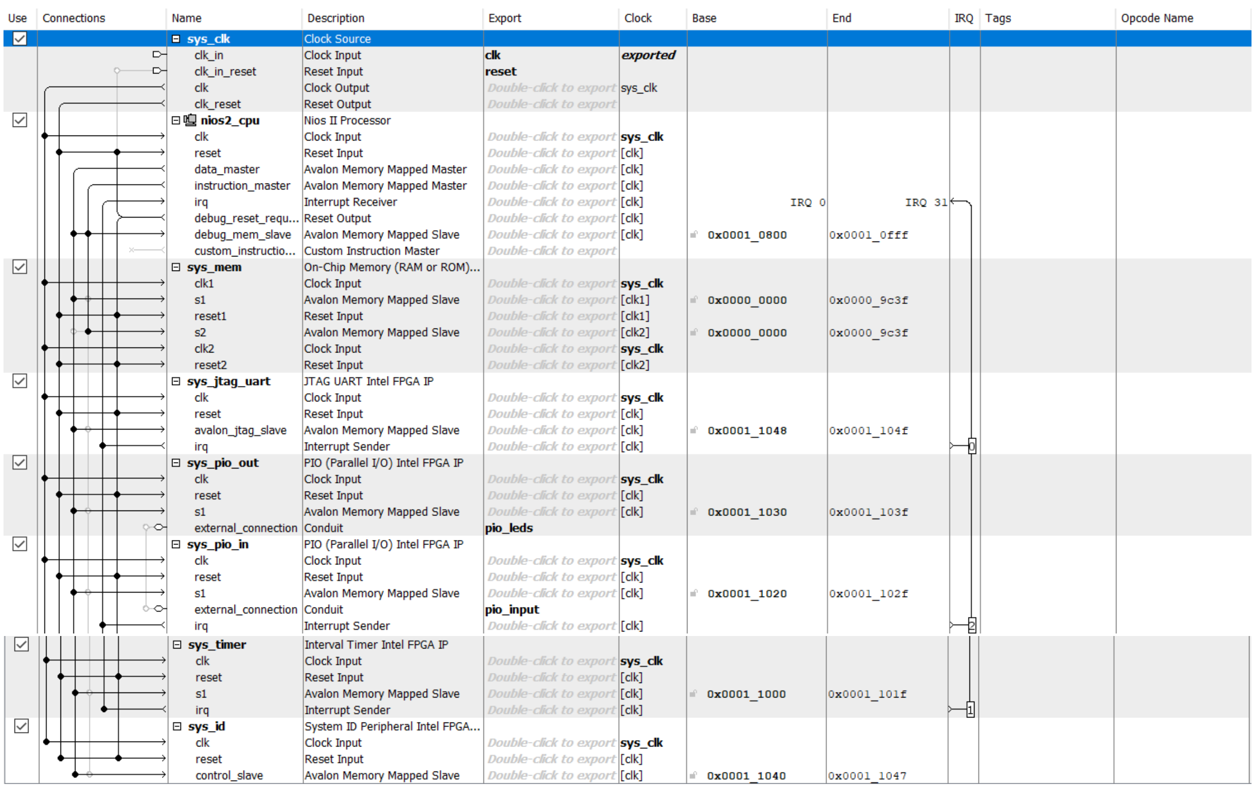
In more detail, SHA-256 is found in block headers to produce unique identifiers for the blocks. Additionally, it is a fundamental component of the PoW consensus, which validates and secures transactions. For that, different users called miners compete to find a valid block hash. In that process, SHA-256 provides the cryptographic foundation for the integrity of the whole blockchain.

2 Part #2: Nios II based SHA-1 implementation

2.1 Nios II Hardware Design

The Nios II design consists of the following components:

- Nios II/f processor core
- On-Chip memory
- JTAG UART core
- System Identification component
- Timer core
- Parallel I/O component for LED control, Parallel I/O component for push-button interfacing



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
		sys_clk	Clock Source							
		clk_in	Clock Input	clk	exported					
		clk_in_reset	Reset Input	reset						
		clk	Clock Output	Double-click to export	sys_clk					
		clk_reset	Reset Output	Double-click to export						
		nios2_cpu	Nios II Processor							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31			
		debug_reset_requ...	Reset Output	Double-click to export	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_0800	0x0001_0fff			
		custom_instructio...	Custom Instruction Master	Double-click to export						
		sys_mem	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	Double-click to export	sys_clk					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0000_0000	0x0000_9c3f			
		reset1	Reset Input	Double-click to export	[clk1]					
		s2	Avalon Memory Mapped Slave	Double-click to export	[clk2]	0x0000_0000	0x0000_9c3f			
		clk2	Clock Input	Double-click to export	sys_clk					
		reset2	Reset Input	Double-click to export	[clk2]					
		sys_jtag_uart	JTAG UART Intel FPGA IP							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1048	0x0001_104f			
		irq	Interrupt Sender	Double-click to export	[clk]			0		
		sys_pio_out	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1030	0x0001_103f			
		external_connection	Conduit	pio_leds						
		sys_pio_in	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1020	0x0001_102f			
		external_connection	Conduit	pio_input						
		irq	Interrupt Sender	Double-click to export	[clk]			2		
		sys_timer	Interval Timer Intel FPGA IP							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1000	0x0001_101f			
		irq	Interrupt Sender	Double-click to export	[clk]			1		
		sys_id	System ID Peripheral Intel FPGA...							
		clk	Clock Input	Double-click to export	sys_clk					
		reset	Reset Input	Double-click to export	[clk]					
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1040	0x0001_1047			

Figure 4 Platform designer Nios II HW design

Figure 4 shows the connections of the IP blocks using the Quartus Prime Platform Designer with its corresponding connections.

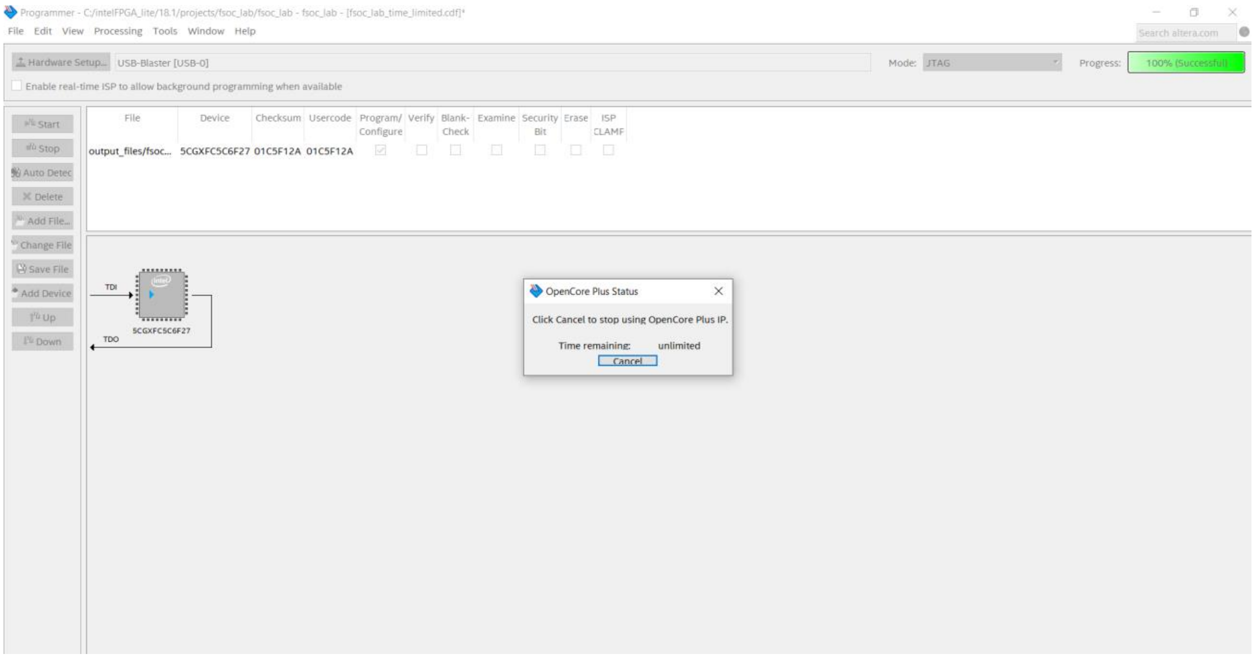


Figure 5 FPGA programming usign Quartus Prime

Once that was done, the fpga was programmed using the programmer tool in Quartus Prime using the sram object file (.sof)

2.2 Top-Level Design and Nios II based SHA-1 Implementation

Once the hardware programming was done, a project was created in the “NIOS II software build tools for eclipse” to allow for the software programming within the NIOS II component that was created.

```
#include "system.h"
#include "sys/alt_stdio.h"
#include "sha1.h"

typedef unsigned int alt_u32;

#define __I volatile const // read-only permission
#define __IO volatile // read/write permission ...
#define __O volatile // write only permission ;-) doesn't work in C...

typedef struct {
    __IO alt_u32 DATA_REG;
    __IO alt_u32 DIRECTION_REG;
    __IO alt_u32 INTERRUPTMASK_REG;
    __IO alt_u32 EDGECAPTURE_REG;
    __O alt_u32 OUTSET_REG;
    __O alt_u32 OUTCLEAR_REG;
} PIO_TYPE;

#define LEDS (*((PIO_TYPE *) 0x80011020 ))
volatile unsigned long delay = 0;

int main(void){
    //message to pass to sha1 function
    char message[] = "FSOC23/24 is fun!";

    // sha1 hash for "FSOC23/24 is fun!"
    uint32_t expectedHash[5] = {0xa617f4b3, 0xa108b6dd, 0x82bb8c4a,
0x16ab0b35, 0x2a32a0b9};
    //hash variable to be used in custom sha1 function
    uint32_t hash[5] = {0};

      //keep track of how many hashes are matching
      int hashMatches = 0;

    //Padded message length calculation (consider that a byte is formed by 8
bits)
    uint64_t padded_messageLen = ((strlen(message) / 55 ) + 1) * 64 ;

    //Message padding
      uint32_t *padded_message = preproces_input(hash, (const uint32_t
*)message);


      for (size_t offset = 0; offset < 64; offset += SHA1_BLOCK_SIZE) {
            const uint32_t *block = padded_message + offset;

            if (offset == 0) {
```

```
                sha_1(hash, (const uint32_t *)block, NULL);
            } else if (offset > 0) {
                sha_1(hash, (const uint32_t *)block, hash);
            }
        }

        //print the hashes
        alt_putstr("Expected hash: ");
        for (int i = 0; i < 5; i++) {
            printf("%08x", expectedHash[i]);
        }
        alt_putchar('\n');
        alt_putstr("Sha1 hash: ");
        for (int i = 0; i < 5; i++) {
            printf("%08x", hash[i]);
        }
        alt_putchar('\n');

        //check if generated hash matches expected hash
        for(int i = 0; i < 5; i++){
            if(hash[i] == expectedHash[i]) {
                hashMatches++;
            }
        }

        // check if hash matches expected hash and turn on or off LED
        if(hashMatches == 5) {
            alt_putstr("Correct hash, turning LEDs ON");
            LEDS.DATA_REG = 0xFF;
        } else {
            alt_putstr("Incorrect hash, turning LEDs OFF");
            LEDS.DATA_REG = 0x00;
        }
        alt_putchar('\n');

        return 0;
}
```

The main file of the base_sys_eval NIOS II ide project has an initial message “FSOC23/24 is fun! Which is sent as the parameter to the custom sha-1 function. If the output hash of the sha-1 function matches the expected hash then the 8 green LEDs of the FPGA will turn on, otherwise the LEDs will turn off.

Figure 6 SHA-1 Nios II based implementation result

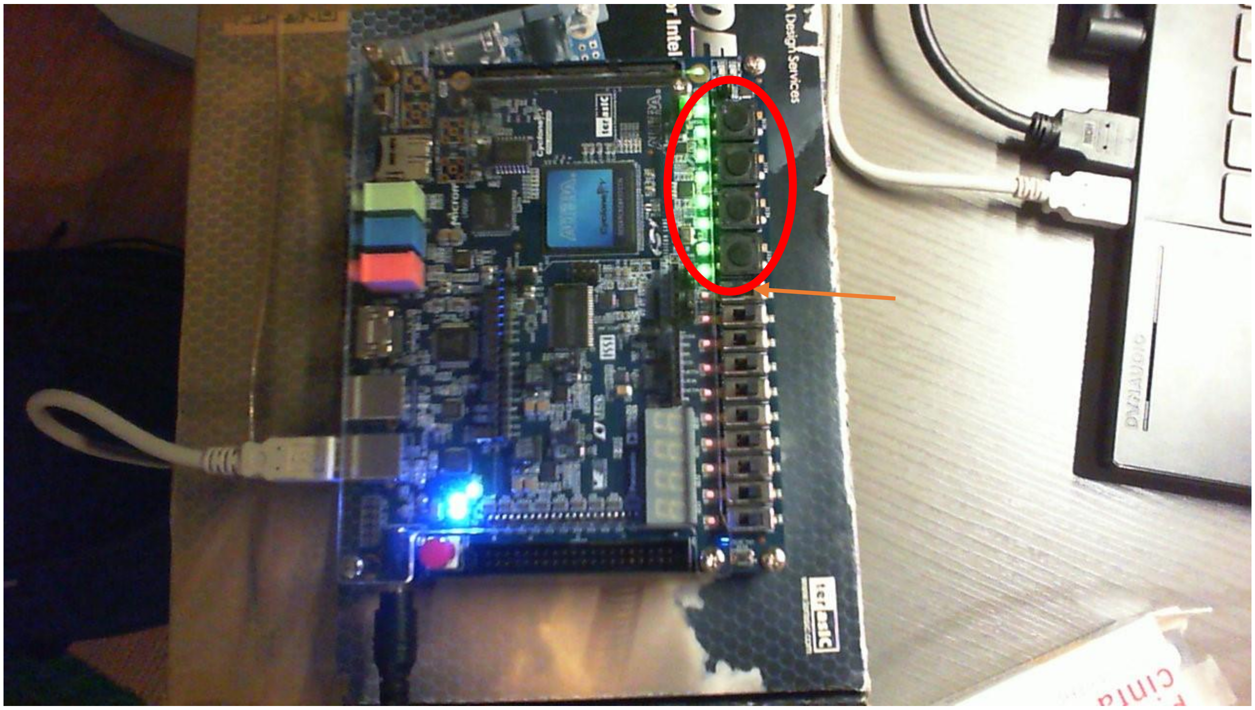


Figure 7 LEDs flashing when correct Hash generated

The console output for the hash being correct as well as the LEDs being turned on in the FPGA. The cpu_reset switch on the board (SW0) must be set high or there is a compiler error.

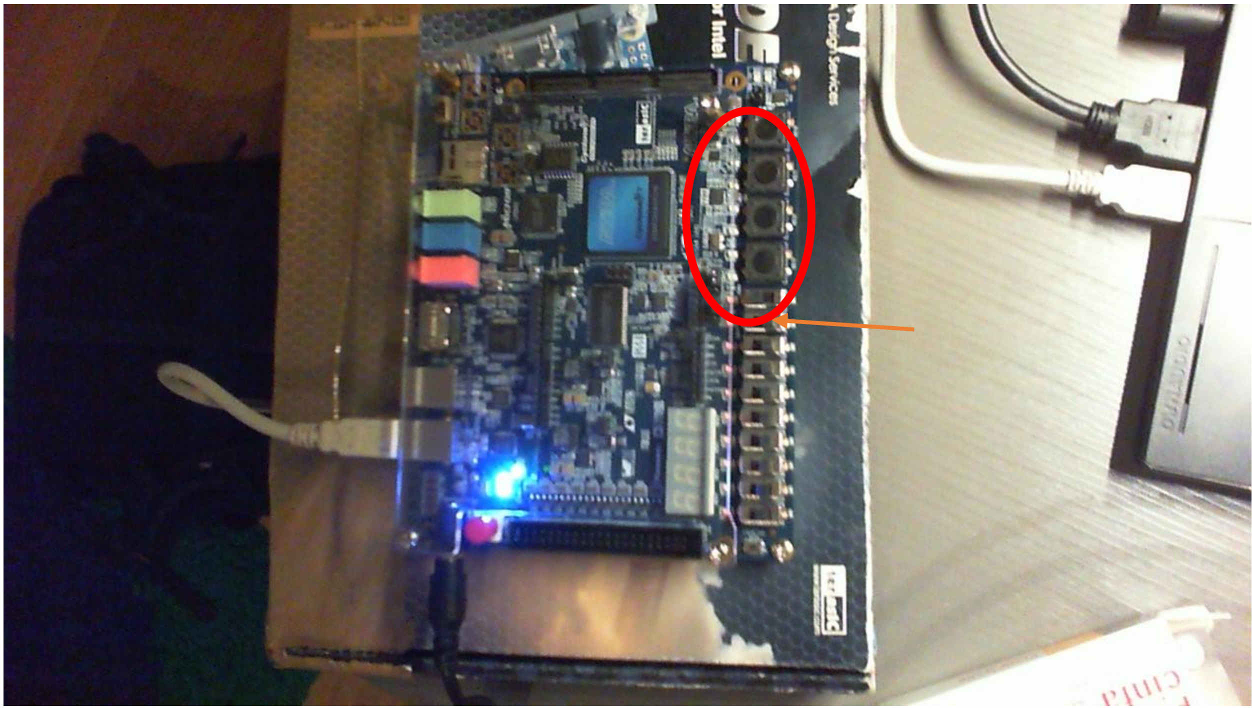


Figure 8 LED's off when Hash is incorrect

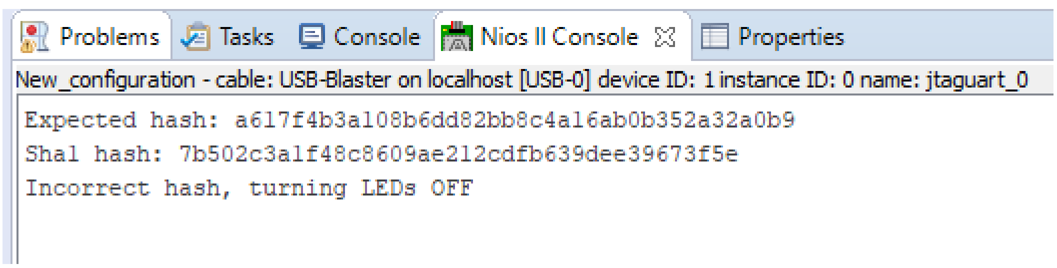


Figure 9 Incorrect hash calculation result

The console output for the hash being incorrect as well as the LEDs being turned off in the FPGA. The cpu_reset switch on the board (SW0) must be set high or there is a compiler error. The message in this case was “Hello World!”