

Implementation & Optimization of the SHA-256 algorithm on a FPGA

Felipe Bichued, Daniel Thielsch

University of Kaiserslautern,
Embedded Systems Group

bichued@gmail.com, d_thiels@cs.uni-kl.de

1 Introduction

The rise of the internet has brought many changes to everyone's daily lives. Next to every application or service, is nowadays available by accessing the web. Even communications are mostly done by applications utilizing the web in some way. To provide such functionalities, most of those services need a way to authenticate users to ensure some sort of security. Even messages or transactions are often authenticated, especially for critical applications. At this point cryptographic methods become a vital aspect for ensuring any security considerations. While there are plenty of different cryptographic methods, most of them use so called cryptographic hash functions. The basic idea of a hash function is to map a variable input message to some fixed length output, named the hash value or digest. With regard to a cryptographic hash function, this hash value or digest can be seen as some sort of *digital fingerprint*[8].

Such fingerprints are particularly useful for checking the integrity of files or messages, after they have been transmitted over a channel. If the fingerprint of the received entity does not equal its original fingerprint, the file must have been changed along the way. Another scenario deals with the storage and handling of user passwords. Instead of storing user passwords in cleartext, only the computed hash value of some password is stored. For authentication, the entered password is then hashed and compared to the stored hash. While there are plenty of possible applications, it must be noted, that most hash functions cannot be considered cryptographic as they do not withstand a cryptanalytic attack. To prevent hash functions from being vulnerable to such attacks, they must fulfill additional properties:

- It must be *one-way*.
Given some hash value v , it must be computationally infeasible to compute a corresponding message m such that $h(m) = v$.
- It must be *weak collision-free*.
Given some message m_1 it must be computationally infeasible to find a message m_2 such that $h(m_1) = h(m_2)$.
- It must be *strong collision-free*.
Finding two messages m_1 and m_2 with the property $h(m_1) = h(m_2)$ must be computationally infeasible.

Looking at those properties it seems fairly logical why each hash function satisfying these constraints is defined to be cryptographic. However actually proving that some given hash function actually has those properties is mostly infeasible. This leads to a situation where the most common cryptographic hash functions are only believed to be cryptographic, as every now and then, new attack schemata against

those functions are discovered. Often these attacks are based on brute force methods, accompanied by some theoretical consideration ¹. Yet most modern cryptographic hash functions are secure for some time to come, so that no attacker can easily tamper with given messages, as every change in the message will yield a different hash value. Though it may seem at first glance, that hash functions are merely theoretical concepts, most cryptographic methods like Digital Signature Standard (DSS), Transport Layer Security (TLS) or even password storage mechanisms guarantee security, by means of some implemented cryptographic hash functions.

2 Problem Description

2.1 Motivation and Context

Since cryptographic hash functions are first and foremost algorithms, they come with some abstract specification, describing how to compute the final hash value. The specification itself is independent of any programming language and can therefore be implemented in any context. This raises the question how an implementation of some specific hash function would perform given some predefined context. And given a straightforward implementation, what optimization might be applied to gain even more performance. Answering some of those questions was the prevalent motivation for this project. The hash function under examination is the *Secure Hash Algorithm 256 (SHA-256)*, a hash function originally published in 2002 by the National Institute of Standards and Technology. It comes along with three additional hash algorithms, namely the SHA-224, SHA-384 and SHA-512, which all together form the 'SHA-2' family of hash functions. As the main difference of all those algorithms is the length of the produced hash value, it is sufficient to consider only one hash function.

For implementing the SHA-256, the programming language *Quartz* was chosen. Quartz belongs to the family of imperative synchronous languages, which work in cycles. Every cycle is determined by reading all available inputs, computing output values based on the current internal state and updating the internal state for the next cycle to come. All actions done during a cycle are considered being instantaneous and do not cost time [1]. Only transitions from one cycle to another, will cost time and are indicated by the small keyword *pause*. Aside from this basic execution scheme, more elaboration on Quartz and its feature set would certainly be beyond the scope of this paper, so that more information might be best found in [5].

In the end, the Quartz implementation of the SHA-256 algorithm will be compiled as to run in hardware on a FPGA[10]. This is mostly done for cryptographic algorithms, as implementations in software tend to be too slow for the required processing speeds. FPGA's are reconfigurable devices, that offer large amount of logic gates and memory, which can be configured by using a hardware description language. The reconfigurability allows for re-engineering even after some specific hardware circuit has been modeled. In consequence it provides a perfect tool for testing different implementations of the same algorithm with regard to performance. The overall goal is to achieve a throughput of up to 1 Gb/s while keeping the latency of algorithm rather low. This performance goal will be the guideline for implementing and optimizing the different variants of the SHA-256 algorithm.

¹See for example the so called *birthday attack*. More information on this attack can be found in [?]

2.2 SHA-256

Having chosen the SHA-256 algorithm as test candidate, a later implementation must adhere to the description provided by the 'Secure Hash Standard' [6]. This description describes an algorithm, which can be separated into two phases, the preprocessing and the hash computation. While the preprocessing is responsible for padding and parsing the message, as well as initializing certain variables, the hash computation itself uses the padded message for deriving a message schedule. Using the message schedule and a host of different functions and variables, a single hash value is computed. This pattern is repeated until the whole message has been read, such that the last hash computation eventually yields the final hash value.

Starting with a rough idea of how the algorithm works, it is now time to look at the details. The first stage of the algorithm is called the preprocessing. Its task is first to read in the whole message data and then pad it according to some scheme. This is done by appending a '1' to the message and a number of k zeros such that the resulting message has a length of $(m + 1 + k) \bmod 512 \equiv 448$. The length m is then encoded into a binary number of length 2^{64} . It is important to notice, that messages of length $m \geq 2^{64}$ cannot be correctly hashed by the SHA-256. Since messages of length $m \geq 2^{64}$ are not very likely, this poses no problem. If in any case such a message should be hashed, SHA-384 and SHA-512 would be the more appropriate hashing functions, as they can handle message lengths of up to 2^{128} . Having padded the message, the resulting message can now be parsed into N 512-bit blocks $M^{(1)}, \dots, M^{(N)}$, while each block $M^{(i)}$ consists of sixteen 32-bit words, $M_0^{(i)}, \dots, M_{15}^{(i)}$. The preprocessing stage is finally concluded by setting the initial hash words $H_0^{(0)}, \dots, H_7^{(0)}$ to some 32-bit constant values, which are then used in the first iteration of the hash computation. After the preprocessing has been completed, the hash computation of all blocks $M^{(i)}$ starts. The input to the i -th iteration of the computation is always the block $M^{(i)}$. Given this $M^{(i)}$, the message schedule/expander derives 64 32-bit blocks W_t , according to the following equations:

$$\begin{aligned}\sigma_0(x) &= ROT_7(x) \oplus ROT_{18}(x) \oplus SHF_3(x) \\ \sigma_1(x) &= ROT_{17}(x) \oplus ROT_{19}(x) \oplus SHF_{10}(x) \\ W_t &= \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases}\end{aligned}$$

where the function $ROT_n(x)$ denotes a circular right rotation of x by n positions and $SHF_n(x)$ denotes a right shifting of x by n position. All operations are done in modulo 2^{32} . Those 32-bit blocks W_t are used for the following iterations of the so called compression function. This function utilizes 8 working variables, labelled a, \dots, h , some 32-bit constants K_t and some intermediary hash value H . For the first iteration of the compression function the working variables are initialized to the hash value of the previous block $H^{(i-1)}$, consisting of $H_0^{(i-1)}, \dots, H_7^{(i-1)}$, while the constants K_t are predefined. Finishing the initialization phase, the compression function is iterated 64-times, to compute a new intermediary hash value $H^{(i)}$. For this purpose the following computations are applied in each iteration:

$$\begin{aligned}T_1 &= h + \sum_1(e)_1 + Ch(e, f, g) + K_t + W_t \\ T_2 &= \sum_0(a) + Maj(a, b, c) \\ h &= g \quad g = f \\ f &= e \quad e = d + T_1\end{aligned}$$

$$d = c \quad c = b$$

$$b = a \quad a = T_1 + T_2$$

where

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0(x) = ROT_2(x) \oplus ROT_{13}(x) \oplus ROT_{22}(x)$$

$$\sum_1(x) = ROT_6(x) \oplus ROT_{11}(x) \oplus ROT_{25}(x)$$

Once the computation finishes after 64 iterations, the new hash value $H^{(i)}$ is simply calculated by using the working variables and the former hash value:

$$H_0^{(i)} = a + H_0^{(i-1)}, H_1^{(i)} = b + H_1^{(i-1)}, \dots, H_7^{(i)} = h + H_7^{(i-1)}$$

While this essentially describes the processing of one 512-bit block of the padded message, the resulting hash value can only be computed repeating the described procedure for all N data blocks $M^{(i)}$. The overall final output is then the last computed hash value $H^{(N)}$.

Besides the SHA-256, the Secure Hash Standard provides a specification for three additional algorithms[6]. Though they produce hashes of different output length, there algorithms look roughly the same. Sharing almost all computation steps with the SHA-224, except the initiation values, the SHA-256 and the remaining hash algorithms SHA-384 and SHA-512 have greater differences. Though having a similar structure, the SHA-384 and SHA-512 do process messages blocks in chunks of 1024-bit, which does not only result in 64-bit operations instead of the original 32-bit ones, but also changes the number of iterations of the compression function as well as some of the underlying shifting and rotation functions. Nevertheless all four algorithms share more characteristics than differences and are thus appropriately called the SHA-2 family.

3 Implementation

Although the specification of the SHA-2 algorithm looks quite complex and intriguing for the first time, a basic implementation can be realised more easily than expected. This is mostly due to the rather sequential nature of the algorithm, where each step mostly depends on previous computations as well to the already supplied interfaces, which make it possible to implement essentially only the algorithm without having to care about a testing environment.

Starting point of the implementation is the interface of the algorithm, which specifies a great number of variables, which are either input or output variables. Logically they can be separated into two parts: The ones necessary for reading in the message and the ones responsible for supplying the correct hash value to the environment. One crucial aspect of synchronous languages is the fact that they allow inputs and outputs to change during the execution of the algorithm. In fact in every cycle the algorithm can work with new input parameters, if they are supplied by the environment. If the environment gives no input value or likewise the algorithm does not produce new output values, the given input and output values are determined by the storage type of the variables. A 'mem' variable will keep its value until the variable value is changed in some future cycle, while a 'event' variable will simply carry the default value of its datatype.

The storage type plays a vital role for reading or outputting the data. For reading in the message, the variable *msg_data* is used, which represents a 32-bit vector and carries the storage type 'mem'. The variable keeps its value, until the algorithm signals the environment to send a new 32-bit word of the overall message. As can be seen, the algorithm does not have access to the whole message right away, but must instead read the message step by step in chunks of 32-bit size. Very important for implementation is the fact, that the algorithm is not allowed to read more than 16 32-bit words, as 16 iteration exactly represent a basic 512-bit block. Reading is only resumed, if a 512-bit block has been processed and the flag *msg_end* is still false. For implementing this behaviour, a *abort statement* has been used, which always checks for every block the number of words read so far.

Things get interesting, when the input flag *msg_end* signals that the whole message has been read, as there are different cases than can occur. If for the number of words k read so far in the current block, $k \leq 13$ holds, then appending the bitvector 0x10000000 allows for at least two additional words, which are needed for appending the length of the input message. As those two words are appended at the end of a block, the bits between $k*32$ and 448 are filled with words consisting only of zeros. It gets more complicated for cases $k \in \{14, 15\}$ and $k = 16$. The first case allows the bitvector 0x10000000 to be appended, but leaves no words for appending the length. As consequence the block must be filled with 32-bit words consisting solely of zeros and a new block must be started, which contains the length as words No. 15 and 16, but is otherwise filled with zeros. In the latter case of $k = 16$, no word can be appended anymore such that again a whole new block must be started which has the bitvector 0x10000000 as first words, followed by words consisting of zero and concluded by the words 15 and 16, representing the length. The challenging part of this implementation is to keep track of all those dependencies whether something has already been appended and if a new block must be started or not. Finding an acceptable and working solution turned out to be actually harder than anticipated.

Regarding the initialization of constants and hash values, which follows the padding of message, this is already done before reading in the first word. Reason for that is that it allows a more ordered code with clean separation of phases like startup, reading and computation. At this place it might be appropriate to point to one of the characteristic features of this basic implementation, namely that all constant values, variables and vectors of the algorithm are locally declared. No external memory constructs or whatsoever are used for retrieving constants or buffering input values. It may not seem like a big deal at first but offloading some of those constants and data to different memory constructs, impacts the overall performance significantly, as will be shown in section 4. While the implementation of the preprocessing turned out to be harder than thought, the actual implementation of the hash computation provides little difficulties.

As the description of the SHA-256 algorithm leaves little room to implement functions or computations elsewhere, all functions have been implemented exactly as described by the specification. For sake of clarity and readability even the identifiers of all working variables and functions are named in correspondence to the specification. Alone the usage of functions might at first glance be irritating, as they are defined as *macro*'s. As most of them are used pretty often during the hash computation, their implementations should be double checked for possible errors. Nothing is more painful than noticing, after searching a couple of hours for eventual bugs, that some shifting operation always shifts by one position less. It is also important to stress that almost all arithmetic operators in Quartz like addition, multiplication and so forth are only applicable to natural numbers. Thus it is necessary to convert all input data, which comes in form of bitvectors to natural numbers, compute the requested operations and convert the result back into a 32-bit vector². Beyond that, there is few worth mentioning, as message schedule

²As for the multiplication, a 64-bit vector is allowed

and message expansion are strictly defined. As the hash computation finishes processing the last 512-bit block, the final hash value is sent in 32-bit words, similar to the initial reading of the message. The whole process of implementing a basic version of the algorithm is concluded by running multiple test cases, as defined by the testing environment. With each test case yielding the expected hash value, the implementation of the algorithm is assumed to be reasonable.

4 Optimization & Performance

So far, the working implementation was only required to compute the right hash values, without any additional constraints. Yet developing a working version of an algorithm and aiming for certain timing or space constraints to be kept, are two different things. While timing or space constraints are more of subsidiary importance in this case, the project goal states a high throughput of up to 1 Gb/s as primary target. It turns out that the basic implementation of the SHA-256 algorithm fails this requirement heavily, as it only provides a throughput of 299 Mb/s³. For increasing this number, different kinds of optimizations have to be applied. As in [4] already stated, one kind of optimization is using special on-board memory, the so called *Block RAM (BRAM)* of the FPGA. This kind of memory allows to offload large amounts of data, like constants or the message data to prefabricated, efficient structures. Using these structures frees up available space on the FPGA, such that more flip-flops or other circuit gates can be used for implementing additional logic. In addition, the extra space allows for an improved routing during the implementation of the circuit and thus in a general speed-up. BRAM's have the nice property of allowing two reads and one write access per cycle, but deliver those values only in the next cycle. Nevertheless the first optimized version of the SHA-256 implementation uses one BRAM for offloading the reads of the message data. As the message is read word by word, the data is immediately written into a BRAM, capable of holding 64 32-bit words. Although every 512-bit block only consists of 16 32-bit words, the expansion phase adds another 48 words, such that 64 entries are just sufficient.

The next 512-bit block then overrides the values the former hash computation, as these 64 words are only dependent on the current input block⁴. Using a BRAM requires to add additional pause statements, as read values are only available in the next cycle. Apart from that, no further changes have to be done, since the actual usage of a BRAM is inferred by the compiling tools, while compiling the Quartz program to run on the FPGA. The impact on throughput for this one optimization is simply striking. As can be seen in figure 1, the column for the first optimized version shows a throughput of 839 Mb/s, roughly an increase by a factor of 2.8. Obviously it makes a huge difference if specialized memory constructs are exploited or not.

Since there is not a single BRAM but multiple ones, the next step could be to use another BRAM for data, which must be accessed quite often and occupies a tremendous amount of internal memory. The second optimized version therefore includes a second BRAM, exclusively for storing the 64 constants K_i , which must all be accessed in every message compression stage. While one normally suggests to first preload the corresponding BRAM and then start with the hash computation, that is not possible in Quartz. As a matter of fact the only workaround is to use an internal array for these constants and then offload them to the BRAM even before the first message data is read in. Of course some performance is lost along this way, but as figure 1 suggests, the throughput of this optimized implementation indeed increases to 972 Mb/s. Though this is no way near the former factor of 2.8, it is enough to come close to the goal of 1

³For details on the definition of throughput see figure 1

⁴This suggests that the intermediary hash values, computed so far, are kept in local registers as they are needed for the next iteration

	Basic Version	Optimized Vers. 1	Optimized Vers. 2	Optimized Vers. 3
Filename:	UserModule	UserModuleDBRAM	UserModuleDKBRAM	UserModuleBRAMUnroll2x
Cycles:	78639	78637	78701	70477
Frequency:	22.489 MHz	62.980 MHz	73.025 MHz	66.278 MHz
Registers:	3,942/20,480 (19%)	1,865/20,480 (9%)	976/20,480 (4%)	1,004/20,480 (4%)
Slices:	2,781/5,120 (54%)	1,124/5,120 (21%)	1,090/5,120 (21%)	1,165/5,120 (22%)
Throughput:	299 Mb/s	839 Mb/s	972 Mb/s	986 Mb/s
Formula Throughput t:	$t = \frac{\text{message length}}{\text{cycles}} * \text{frequency}$			

Figure 1: Comparison of the different implementations of the SHA-256

Gb/s throughput. Another optimization approach provided by [4], suggests a technique called unrolling. While the cited unrolling mostly concerns the architecture of the combinational logic, the corresponding software technique is called *loop unrolling* and is considered to be a compiler technique. The idea is to replace an entire loop by inserting for every iteration the loop body instead. Especially for small loops this yields some performance gains as the compiler can optimize the code without considering any loop barriers. For large loops it might also be preferable, if space requirements do not effect the performance. But as was shown, this does not hold for the SHA-256 implementations, were most of the speedup was gained by using as less local memory as possible.

To cope with this problem, a third optimized version was developed, which instead of unrolling the entire message compression stage, settles on doing two iterations in its loop body instead of one. And surprisingly additional throughput can be gained by this small change as the throughput increases to now 986 Mb/s, see figure 1. A somewhat small change, but nevertheless an increase in throughput and again proof of the fact that most concepts are useful if they are tailored to the requirements. Another interesting aspect can be seen, if cycles and cycle frequency of the second and third optimized version are compared. It shows that cycles and cycle frequency are negatively correlated. Since both versions are optimized pretty much, the amount of work to be done is roughly the same. Consequently spending less time in one cycle, leads to a higher frequency but must be compensated by more cycles in total. Whereas deep cycles, which do a lot of work, lead to fewer cycles and decreased frequency. Both versions seem highly optimized as they only trade cycles for frequency and do not show large variations in terms of throughput. A listing of all the different versions and their characteristics can be seen in figure 1.

5 Conclusion

In summary it can be said that the project and its underlying goals made for a great learning experience. At the beginning, working and developing with the synchronous language did not seem complicated but the rich feature set of the language offers some subtleties, so that same errors are inevitable. Especially figuring out how to come up with an solution for the different cases of padding the input message, proved harder than thought. Still, considering this delays, the optimized versions came pretty close in matching the specified target throughput of 1 Gb/s, such that this goal was for the most part achieved. Moreover it proved important to keep ones target hardware in mind, while developing the algorithm, as most performance gains were largely caused by exploiting certain memory constructs provided by the target platform. This is a lesson to keep in mind, especially in the embedded domain, but also with respect to general software engineering. While there are certainly some optimizations left, the next step would very likely involve diving into the circuit and changing architectural assets were necessary. Even the tools, compiling the Quartz code to the FPGA might offer some means for adjustments, but as their complexity sometimes nearly overwhelmed us, this is best left to others for exploiting.

References

- [1] (2013): *Quartz Introduction*. Available at <http://www.averest.org/documentation/Quartz-Introduction.pdf>.
- [2] (2013): *Quartz Reference Card*. Available at <http://www.averest.org/documentation/reference-card.pdf>.
- [3] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa & Stamatis Vassiliadis (2006): *Improving SHA-2 Hardware Implementations*. In: *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Lecture Notes in Computer Science 4249*, Springer, pp. 298–310. Available at <http://www.iacr.org/cryptodb/archive/2006/CHES/24/24.pdf>.
- [4] Robert P. McEvoy, F.M. Crowe, C.C. Murphy & William P. Marnane (2006): *Optimisation of the SHA-2 family of hash functions on FPGAs*. In: *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 00, pp. 6 pp.–.
- [5] K. Schneider (2013): *The Synchronous Programming Language Quartz*. Available at <http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf>.
- [6] National institute of standards and technology (2002): *FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2*. Technical Report, DEPARTMENT OF COMMERCE. Available at <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>.
- [7] The Free Encyclopedia Wikipedia (2013): *Birthday Attack*. Available at http://en.wikipedia.org/wiki/Birthday_attack.
- [8] The Free Encyclopedia Wikipedia (2013): *Cryptographic Hash Function*. Available at https://en.wikipedia.org/wiki/Cryptographic_hash_function.
- [9] The Free Encyclopedia Wikipedia (2013): *SHA-2*. Available at <http://en.wikipedia.org/wiki/SHA-2>.
- [10] Xilinx (2013): *XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*. Available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/xst.pdf.