SHA-256 FPGA Implementation and Benchmarking

Asatur Marzvanyan, Akashdeep Takhar, Bryan Arciniega, Connor Carr, Emanuel Urbina, Mohamed El-Hadedy

dept. Electrical and Computer Engineering
California State University Polytechnic, Pomona

***Abstract*—Performance metrics allow us to determine several attributes of a given product. In terms of software implementation upon a hardware device, it is possible to record measurements of instruction speed and clock cycles to observe the efficiency of the code implementation. A software field in which many people regard efficiency as an important component in terms of security is in cryptography. To ensure the best security for password and other important private information, it is best to practice securing the information at a fast speed and to determine the number of cycles the implementation will take to encrypt it. A weak cryptographic algorithm allows for more vulnerability and exposure to secure information. To handle this risk, it is in best practice to measure benchmarks for cryptographic implementation upon hardware. Benchmark data is taken to measure certain important characteristics of a project to determine the performance. In the case of cryptographies, benchmarks include CPI, MIPS, Clock Cycles, and types of processors the securing code is implemented upon. We decided to implement the SHA-2 cryptographic hash function on the Xilinx Nexys A7 field programmable gate array to observe the code and gather benchmark data. We all have experience with using this FPGA and understand the process for implementing C code within Xilinx Vitis. With this data we can compare the SHA-2 code upon each member's personal computer as well to observe the speed on different processors. Benchmark data collected in this study allowed for us to better understand software implementation upon hardware.**

***Index Terms*—Algorithm, Authentication, Benchmarking, Bits, CPI, Cryptographic, FPGA, Hash, Microblaze, Nexys A7, Optimization, SHA-2, SHA-256**

I. Introduction

A specific cryptographic algorithm that we will examine is called the SHA-2 (Secure Hash Algorithm 2). This secure hashing algorithm receives an input and hashes out a unique and secured output that is 256 bits long through a hash function. The SHA-2 is the most reputable securing hash algorithm that many companies prefer to use over the previously cracked hashing algorithms such as the SHA-1 and MD5[1]. SHA-1 was capable of only producing a 160-bit hash output was reportedly cracked in 2017. With signs of further weakness and increased vulnerability many people decided to move on to a more secure algorithm. Since then, SHA-2 is the most secure algorithm to use for encryption of private data and information. This secure hashing algorithm was established in 2001 and developed throughout the years by the National Security Agency with the goal of providing the most secure algorithm that people may use[2]. With this unique cryptographic code, we began to question its efficiency on different types of hardware[3]. The execution and runtime of the code may differ on different computers because of the computer architecture the different components used to build the computer together. Although the code itself is great for hashing information, we assume that execution of this secure hashing algorithm might differ on different microprocessors. For us to experiment with running this code and observing its efficiency, we decided it would be best practice to collect benchmark data on different microprocessor to note not only differences in performance, but also the behavior of the computer as it runs the SHA-2 Algorithm.

II. Hash Algorithm

A hash algorithm is an algorithm in which an input is taken into a hash function and a string of characters is produced which encrypts the original input message. Hash algorithm are unique because they are difficult to trace back in order to find the original input data. Without information of the input data, it is quite difficult to decrypt the hash output. In the case of SHA-2, the hash algorithm is more complex to ensure that the input information is nearly impossible to obtain unless you are given the input data.

Algorithm 1 is used from the US Secure Hash Algorithm [4] as derivation for our own custom implementation for the MicroBlaze microprocessor and it is compared to another software implementation from a third party source using C++ in visual studio [5]. This third party source was modified and implemented on a few different microprocessors, Operating systems, and even architectures. The thrid party source was implemented on windows 10, linux, and mac os. Windows 10 and Linux were both based on various microprocessors ranging laptop grade intel, old generation desktop intel and recent AMD Ryzen microprocessors. Our Mac OS however, is equipped with apples new M1 microprocessor running on an ARM based architecture and will show us some interesting results.

III. Benchmarks

To measure performance and observation of the SHA-2 upon different microprocessors we collectively decided to use specific benchmarks to record data. To begin this experiment, we recorded data of the SHA-2 upon different computers of those involved in this research, as well as on a virtual microprocessor on a FPGA program called Vitis. Vitis provides a microprocessor called Microblaze, which is compatible to run upon a Nexys A7 field programming gate array board. We recorded execution time to determine the speed at which microprocessors execute the SHA-2 code. In order to observe additional behavior of the microprocessor we recorded the instruction count and cycles per instruction.

Algorithm 1 SHA-256 Processing

```
procedure PREPARE MESSAGE SCHEDULE W:()
    For t = 0 to 15
      Wt = M(i)t
    For t = 16 to 63
      Wt = SSIG1(W(t-2)) + W(t-7) + SSIG0(w(t-15)) + W(t-16)
end procedure
procedure INITIALIZE WORKING VARIABLES:()
    a = H(i-1)0
    b = H(i-1)1
    c = H(i-1)2
    d = H(i-1)3
    e = H(i-1)4
    f = H(i-1)5
    g = H(i-1)6
    h = H(i-1)7
end procedure
procedure PERFORM MAIN HASH COMPUTATION:()
    For t = 0 to 63
      T1 = h + BSIG1(e) + CH(e,f,g) + Kt + Wt
      T2 = BSIG0(a) + MAJ(a,b,c)
      h = g
      g = f
      f = e
      e = d + T1
      d = c
      c = b
      b = a
      a = T1 + T2
end procedure
procedure COMPUTE INTERMEDIATE HASH VALUE()
    H(i)0 = a + H(i-1)0
    H(i)1 = b + H(i-1)1
    H(i)2 = c + H(i-1)2
    H(i)3 = d + H(i-1)3
    H(i)4 = e + H(i-1)4
    H(i)5 = f + H(i-1)5
    H(i)6 = g + H(i-1)6
    H(i)7 = h + H(i-1)7
end procedure
```

Since the microprocessor executes instructions of the code we decided to observe the number of cycles that the microprocessor must perform per instruction and in addition to this we also came to the conclusion that counting the number of instructions will determine the behavior of the SHA-2 upon these microprocessors. Along with these benchmarks we also recorded the millions of instructions per second to further observe the time and instruction behavior in the microprocessor.

IV. Methodology

This project required a SHA-256 implementation that could run on both a general-purpose consumer computer with an Intel-i7 processor and a Microblaze processor on the Aritx-7 FPGA board. The Vitis/Xilinx toolchain limited the available development languages to C and C++; ultimately, in the interest of easing development, we selected C++ for our implementation. Our implementation requires a byte array as input data and a value indicating the length of data to hash. This value cannot be larger than the input array or the result will be undefined. The word size is required to be 32bits long and must obey modular arithmetic, which led us to use unsigned 32bit integers as our word type.

The implementation works by copying some of the input data into a 512bit block, then performing the hashing function on it, and then setting up the next block to be processed and repeating the cycle until all the blocks have been processed. In the case of the last block, the bit following the end of the data is set to one, and then padded with zeros until reaching the last two words of the block which are required to contain the 64bit size of the input data.[6] The hashing functionality uses an eight-word buffer to hold the result hash values, this buffer is initialized to a specific set of hex values that are derived from the fractional parts of taking the square root of the first eight prime numbers.[4] The process for hashing a single block starts by setting up a 64-word buffer called the message-schedule, the first 16 words consist of the words in the block, and the rest are initialized to a value resulting from an expression that adds, shifts, and rotates values previously initialized in the message schedule. Then a set of eight working-variables, referred to individually by the SHA-256 standard using the letters a-g, are initialized to the current values held in the result buffer. The message-schedule is then iterated through, with each loop modifying the working-variables using a specific arithmetic expression primarily consisting of addition and bitwise rotation. Following this, the contents of the working variables is added word-wise to the results buffer. At the end of the hashing process for the entire message, the result buffer contains the sum of the working variables from each individual block's processing in addition to its initial value.

To profile this program, we used separate tools for each platform to get the measurements needed. To take measurements on a general-purpose desktop, we used the Linux-based Perf tool. This tool is purpose-built to record a variety of data during a program's execution through the use of hardware performance counters.[4] This tool allowed us to determine the number of instructions executed, the number of clock cycles throughout the execution, the speed of the processor, and the execution time.

To profile the Microblaze, we had to employ a variety of methods. To determine the number of executed instructions, we created a hardware platform that had a Microblaze processor with debugging enabled. Our debugging options were limited, but we were able to create a Tcl script that used the debugger to step through the entirety of the program one instruction at a time. This process was able to tell us the number of instructions that were executed. We were able to determine that certain parts of the program run in the same number of instructions regardless of the input, and were able to speed up the debugging process significantly. To measure the execution time and number of clock cycles,

we made use of a hardware timer attached to the Microblaze processor. This timer was designed in Verilog to interoperate with the Microblaze processor via an IO bridge and a driver that allowed programmatic control from code executed on the Microblaze processor.

V. Results

The results to our testing were interesting to compare as there were a lot of differences in performance on different platforms. The most compelling one being the amount of instructions as well as the amount of clock cycles completed, these two being most affected by the different implementations. When looking at the hardware implementation, the clock cycle frequency was constant, as it was a fixed variable that could not be changed, making it perfect for comparing to software implementation, however, the amount of instructions varied greatly from both platforms, in turn, changing the amount of clock cycles that both platforms had to go through before the program would finish. Looking at the one of our results, we can see that, in general, the hardware implementation had a higher CPI (clock cycle per instruction) as it averaged a 1.78 whereas the software implementation had a lower average CPI at 1.42. This, however, is not the whole answer, as the amount of time it took the Microblaze to execute its function was almost 3 to 4 times slower in its execution time, taking an average of 2159.1 microseconds for the program to execute while the software implementation only took an average of 551.6 microseconds to execute. One reason for this could be that while the time was faster, it also had a lot more clock cycles to go through as well as larger amount of instructions, allowing a more streamlined process. While this makes it seem like the Microblaze would be a worse implementation, this is certainly not true, as this part of our results only focused on a certain part of the testing. When looking at just the times for our code implementation (reference to redo of our code), it shows that the Microblaze had been faster than some processors that we had run, and while it was not possible to find the instruction count with these results, it does show that the Microblaze implementation has more potential.

Averages	MBlz(abc)	MBlz(cstm)	Intel(abc)	Intel(cstm)
Clockspeed	100Mhz	100Mhz	3.076Ghz	2.87Ghz
Exec Time	1420us	2898us	549us	554us
ClkCycles	140047	287628	853153	855165
Instructions	79094	160584	580188	622625
CPI	1.771	1.791	1.488	1.374
MIPS	55.69	55.41	1057	1124

TABLE I: Micro Blaze Implementation vs. Linux Implementation

The data presented in Table I we can see that we were able to record accurate results for our MicroBlaze and Linux Implementations. This allowed us to calculate the CPI and the MIPS to compare. The MicroBlaze being a hardware implementation we can see it benefits from a reduced instruction count but because it is still limited to the 100Mhz clock speed of the A7 we see slower but still reasonable execution times. Looking at the Linux implementation we can confidently see the how the instruction count is substantially increased to compensate the use of a Linux based compiler.

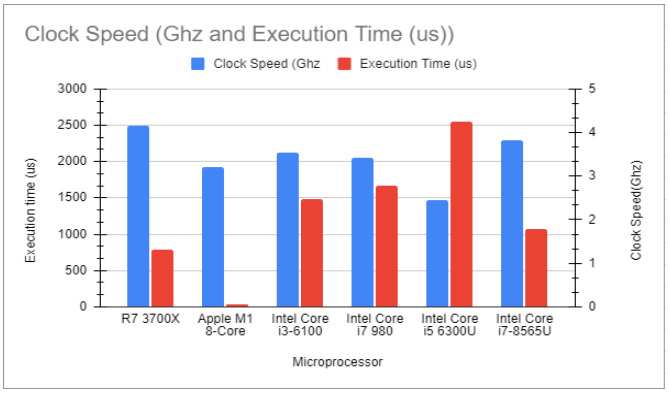

Fig. 1: Bar graph of our data

Student	Processor	AVG.ClkSpeed	AVG.ExecTime
Asatur	R7 3700X	4.168Ghz	783us
Bryan	i3 6100	3.53Ghz	1483us
Emanuel	i7 980	3.43Ghz	1669us
Connor	i7 8565U	3.82Ghz	1077us
Akash	i5 6300U	2.44Ghz	2544us
Bryan	Apple M1	3.2Ghz*	39us

TABLE II: Windows 10 and Apple M1 Execution Time of third party code

Table II was tricky to work around but specifically we could not reliably get any other data other than execution time, To clarify the Execution time represented here is the specific execution time of the Hashing algorithm instead of the whole code. This is because windows 10 does not allow us to read instruction count through methods like program counters, that were used in other implementations, for security reasons. Visual studio both could provide us with the ASM code which would provide a static count of Instruction and our execution time, However this was not an accurate method as the instruction count will be varied based on the message/string used to hash, the looped instructions that a static instruction count can not tell us, and the general inaccuracy of the execution time that was reported. We would need to be able to read the Dynamic instruction count which just was not possible on windows 10. Instead we modified the third party code to include a new Chrono Library and ran a high resolution clock at the start and end of the portion of code that handled the hashing algorithm. With this we were able to get our execution time much more accurately than what visual studio could provide us. Unfortunately we were not able to obtain any more data from Windows 10 but the new Apple M1 processor tells us a different story.

Processor	B.Clk Speed	Inscrution Count	# of Cycles	CPI	MIPS
Apple M1	3.2Ghz	4,621,647	2,456,476	0.532	200.94

TABLE III: Apple M1 data

The new Apple M1 is an ARM based Microprocessor designed on a 5nm process with over 16 billion transistors [7].

Being on the bleeding edge of technology and implementing an ARM architecture definitely has its benefits. ARM is a RISC machine, also called "Load/Store" type architecture. All ARM instructions are 32 bits long and most of them execute in one clock cycle [8]. This approach with RISC is one of the key reason we are seeing such a massive performance difference between architectures. The x86 architecture is designed for general purpose computing and has its own set of benefits over RISC like having a ton of instruction sets implemented but this also applies some overhead to the whole process.

Taking a look at the CPI and MIPS generated by the M1 microprocessor that our CPI is considerably lower at almost half a clock per instruction which is much better than the other implementations, however the MIPS are lower when compared to the Linux implementation. Looking deeper we can see that the ARM based microprocessor takes overall less cycles for the amount of instructions which explains why the CPI is much lower but the increase of the instruction count because of the RISC based architecture is why we see a lower MIPS count overall.

VI. Conclusion

In this experiment we used two C-based source code programs to implement and benchmark a SHA-256 algorithm on a FPGA board namely the Artix A7 100T. The results indicated above show that the implementation of the SHA-256 algorithm was much faster than its x86 counterpart. The reasoning behind the drastic increase in performance is because of the FPGAs fewer abstraction levels compared to the microprocessor in the computer. However, there was a comparative difference when it came to Apple's new M1 arm-based chip. We saw that the M1 performed magnitudes faster execution time compared to the x86 computers. Although execution time is not a complete measurement of superiority since we could not accurately get the instruction count for Windows computers, we believe it is a good indicator of performance. We also saw the CPI and MIPS were much more favorable to the M1, 0.53 and 200.94 respectively. This research is prescient given that hashing is becoming much more embedded into consumer's everyday lives as privacy concerns are increasing and cryptocurrencies has begun to go mainstream. For this reason we believe further research into the implementation of SHA-256 on FPGA could result in greater hashing throughput than general purpose computing while still being reprogrammable compared to ASIC.

References

[1] K. K. Ting, S. C. Yuen, K.-H. Lee, and P. H. Leong, "An fpga based sha-256 processor," in *International Conference on Field Programmable Logic and Applications*. Springer, 2002, pp. 577–585.

[2] M. Khalil, M. Nazrin, and Y. Hau, "Implementation of sha-2 hash function for a digital signature system-on-chip in fpga," in *2008 International Conference on Electronic Design*. IEEE, 2008, pp. 1–6.

[3] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, "Optimisation of the sha-2 family of hash functions on fpgas," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*. IEEE, 2006, pp. 6–pp.

[4] D. E. 3rd and T. Hansen, "Us secure hash algorithms (sha and sha-based hmac and hkdf)," *RFC 6234*, May, 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6234

[5] hak8or. (Feb, 2014) Sha-256 basic implementation in c++ with a test. [Online]. Available: https://gist.github.com/hak8or/8794351

[6] N. I. of Standards and Technology. (August, 2015) Secure hash standard (shs). [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.180-4

[7] Apple. (November 2020) Apple m1 chip. [Online]. Available: https://www.apple.com/mac/m1/

[8] T. Neagoe, E. Karjala, and L. Banica, "Why arm processors are the best choice for embedded low-power applications?" in *2010 IEEE 16th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2010, pp. 253–258.