Hardware Acceleration of MD5 Algorithm for Blockchain Applications

Ashley Ly
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
lyashley@usc.edu

Nikhil Sinha
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
nikhils@usc.edu

Richmond Johnson David Bhaskaran
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
davidbha@usc.edu

***Abstract*—Blockchains are becoming an increasingly popular tool for cryptocurrency, inventory management and tracking in the recent days. Security is one of the key advantages of using Blockchains, as every node in the chain assures the integrity of its data as well as that of its neighbors by means of hash function calculation. This paper discusses the generation of MD5 algorithm along with a pipelined implementation of the same and its associated modules. By identifying the memory access stages and pre-calculating the values to be used in the hash generation, along with forwarding help between the stages, the time slice per stage is greatly reduced, thus promising fast and efficient generation of the hash over many input blocks of data. This hardware accelerator can not only be used in message transmission, but its message digest properties can be used in dynamic evaluation of data in embedded systems and digital displays content evaluation.**

I. Introduction and Motivation

There are many applications today that hashing algorithms are applied to. Hash chains are long sequences of hashes that use the output of one hash as an input to the next hash, along with the data of each step. This allows for is linking of data from one segment to another in a fashion such that order is verifiable. When another party wishes to know if the data that they get is valid, they can check the hash chain by running their data through it at each step and seeing if the hash chain values are the same. This concept is used in multiple applications, such as blockchain, data correctness, password protections and network authentication. [3] uses the hash chain to initialize microtransactions using long sequences. Through these long hash chains, a greater level of security and assurance can be attained. However, because of the length of the hash chains and the complexity of the hash functions, the process of generating a hash chain can take a long time. Therefore, the goal of our project is to decrease the amount of time that it takes for the entire hash chain to be generated, allowing for quicker use of the applications specified before.

II. Previous Works

Hardware acceleration can help with poor performance due to increased complexity from the cryptographic algorithms used in the SSL/TLS protocol. In [2], a FPGA-based VLSI Crypto-System was proposed for embedded applications that need high security, speed, and low power consumption. The cryptographic functions: Scalable Encryption Algorithm (SEA), Message Digest Algorithm (MD5), and Secure Hash Algorithm (SHA-2) were integrated into the accelerator core for use in networking security through the OpenSSL library. Performance analysis measured the area, execution time, and power consumption of the proposed system. [2] concludes that hardware acceleration improves the performance of computation intensive cryptographic algorithms for SEA, Message Digest Algorithm (MD5), and Secure Hash Algorithm (SHA-2) in terms of speed, optimized area, and enhanced level security for the target Cybernetic application. The authors of [2] look to include low power ASIC implementations of Quantum SSL (QSSL). However, [1] and [2] both used FPGA-based systems in their explorations for their balance between reconfigurability and hardware acceleration. FPGA-based systems are the best platform to explore our own approach of hardware acceleration in cryptographic algorithms.

The MD5 algorithm is a relatively straightforward algorithm that provides a one to one source to

destination message mapping. It involves several shifting and logical operations, and the only downside is the sequential relation between the various stages. The message is broken into integral number of 512-bit sections with only the last section containing the overhead such as the message length. Every 512-bit section is further divided into 16 numbers of 32-bit sections. The 512-bit section will then go along a repeatedly executing 64 step process involving four buffers. The approach cited in [4] has taken the repetitive equation mentioned below and has tried to divide it into hardware slices. The equation under consideration is

$A = B + ((A+Func(B,C,D)+X[k]+T[i])<<< s)$ ->(1)
$A \leftarrow D; B \leftarrow A; C \leftarrow B; D \leftarrow C$ -> (2)

The paper tries to resolve the above two equations into individual pipeline stages and provide forwarding data between the various stages. The equation is broken into elementary steps and three pipeline options are being explored – 3 stage pipeline, 4 stage pipeline and 3 stage pipeline with BRAM. The BRAM is used to store the initial values of the buffers, the constant T[i] table and the intermediary values. Additionally, the buffer contents are fetched in advance during the pipeline stages themselves as they will not be used further down the pipe. The pipeline design tries to concentrate the bulk of the Func(B, C, D) and the shifter logic to the final stages, which takes the majority of time slices. Thus, the proposed design tries to achieve higher computational speed of the algorithm by promoting hardware acceleration and potentially, it is one of the few areas where parallelization can be implemented as well as the core logic of the MD5 algorithm.

In addition to hardware acceleration of hash algorithms, our approach also looked at the possibility of using parallelism to improve the speed of hash algorithms. [3] explores the effects of pipelining on the implementation of MD5. The findings of [3] indicate that pipelining reduces both delay and area requirements when a single MD5 step is observed. However, the overall area requirements increase as a function of the number of pipelined stages. 2- and 4-stage pipelining increase the area requirements only moderately while providing faster performance. 32-stage pipelining resulted in considerably larger area requirements with negligible benefits. Full pipelined architectures result in area requirements so high that it can only be recommended where very high performance is demanded.

Our approach relies heavily upon the maximization of throughput for hash chains. [3] presents multiple ways to implement the MD5 algorithm in hardware, as well as throughput maximization techniques using pipelining. As [3] states, “The structure of the MD5 algorithm allows both iterative and pipelined implementations”. This is extremely critical to our project as we are reliant upon this algorithm to decompose for hash chaining purposes. The paper aims to minimize the time taken to do initialization for a hash chain of length 10,000, which is extremely similar to the reduced blockchain problem we will present in the next section. The way that the paper approaches pipelining is to break the MD5 hashing algorithm into its core mathematical steps. It then makes some or each of these steps into stages, trying out different pipelines and showing the results. The authors find that there is a tradeoff between the performance and the size of the logic as they decompose the algorithm further and add more stages to the pipeline. The paper also focuses on improving the speed of a single hash using a hardware accelerator as well by implementing the hashing procedure in Verilog, essentially a one stage pipelined version of their work. Their results show that hardware implementations significantly speed up the overall time to hash 10,000 elements, and that the performance and throughput improves as the number of pipelined stages grows.

III. Approach

Our approach is to use the ideas brought forward in pipelining to create a system that is faster to hash many blockchain elements than a traditional system. Current systems require lots of hashing time as there are multiple different hashes that need to be calculated for every step in the blockchain. First, a block must calculate its own hash along with the hash of the previous element. Then the transaction needs to be signed with the sender’s private key to validate the transaction. For the purposes of our project, we will leave out this portion of the blockchain process and focus only on the high volume of hashes necessary when creating or validating a large blockchain. During validation, a node on the chain will run through every

block in the chain and calculate its hash using the sender's public key. This allows them to know for every block whether it has been signed by the sender and therefore is valid.

Software Benchmarking:

For phase 1 of our project, we wanted to get a good baseline as to what the timing for a software only solution would be. To do this, we developed a reduced model of a blockchain to run testing on to get an accurate representation of how current software libraries deal with hashing on a large scale. We use the base library hashlib that is standard in python and the MD5 hashing algorithm to do all the hashing in our model. Our model is based on the code at https://medium.com/crypto-currently/lets-build-the-tiniest-blockchain-e70965a248b where the author creates a basic model of a blockchain without too much of the digital signature involvement. What this model encapsulates is the many hashes that are necessary for the chain to be populated, as at each step we chain together blocks by hashing their hash along with the current block's data.

Our model is split into two main parts, the block and the chain. The block takes in data, timestamp and the hash of the previous block and generates its own hash, all of which it stores internally. The chain is implemented as a simple array that holds all the blocks in order. Therefore, to validate this chain, one must go through every block and hash the block's data and the hash of the previous block to see if the current block has a valid hash. To test our model, we generate 100,000 random data sequences of any length between 1 and 100 to function as the data inputs to our blocks. We then create a block for each data sequence and add that block to our chain. Overall, we must perform 100,000 hashes to add all the blocks to the chain successfully. We also print the data and hash for each block to a file so that we can later verify that the process did generate a hash. Our output file looks like the following:

From running this process 100 times and collecting the times for each run, we have concluded that it takes on average 3.1467 seconds to add 100,000 blocks to the hash. While that may not seem like a large amount of time, this is only a subset of the blockchain procedure. Our result implies that each hash is taking 31,467 nanoseconds to hash. On a system running at a 20ns frequency, this is around 1600 clocks per hash. We improve on this using our hardware design.

Hardware Implementation:

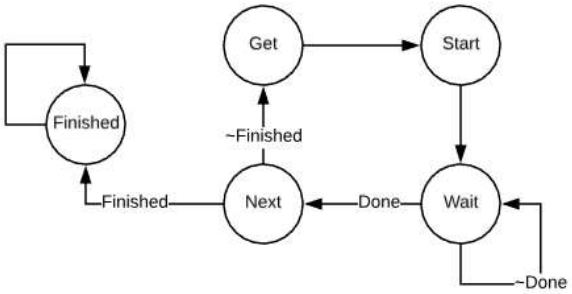
The hardware implementation, deals with two major design ideas - To improve generation of hash within a single round and to improve the generation of hash over multiple rounds. The first challenge can be handled by implementing a dedicated MD5-Core module that will work over only one round, using a limited set of resources. This unit will be pipelined such that all the data necessary to do the calculation as mentioned in [2] will be pre-fetched, pre-calculated and only the stages that cannot be further subdivided are merged into a single stage. To handle second challenge, there should be a top module which will take care of the overhead functions in a MD5 hash generation procedure such as fetching data from the ROM, calculation of the message length, segregation into 512-bit block and further segregating a block into 16-word blocks of 32 bits. In addition to the above, it should also govern the currently processed block and should also make decisions to further the process or stop the process and

generate the output Hash value, intra-round values to be exchanged and the like.

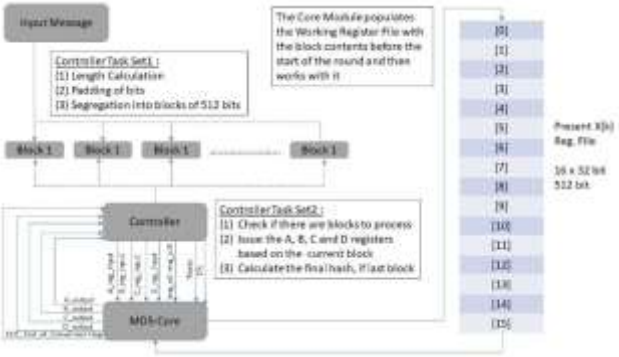
Therefore, the data flow is as follows: the input message is stored in the ROM as a single entity. The Controller module will fetch the input data and will perform the overhead functions. Depending on the message length, it will select one 512-bit block at a time, segregate into sixteen 32-bit words and supply it to the Core module along with the initial A, B, C, D register contents depending on the round number. The MD5-Core module which is a pipelined hardware accelerator will first populate the Register File of the MD5_Core module before the start of the round and then work on the current 512-bit data block and will generate four 32-bit words sent back to the controller module. The controller module, depending on whether there is any more data left to process or if it is the final round, will accordingly populate the Register file and will perform intra-round operations in preparation for the next round or will perform the final round calculations and generate the hash output respectively.

Controller Module:

To perform hash calculation of a chain, we have developed a controller to orchestrate the hashing process for the MD5 hash chain. The controller module interfaces with the ROM and the MD5 core. First, the data is loaded into the ROM. Then the controller goes through each data segment in the ROM and brings it into the controller. It then concatenates the value with the result of the hash from the previous segment of data, or 0 for the beginning of the chain, and sets that value to the MD5 core to be worked on. The controller then activates and deactivates the reset signal of the MD5 core, triggering the MD5 core to calculate the hash of the next block. The controller waits for the MD5 core to be complete and then captures the result of the hash from the core module. It then begins the process again by grabbing the next segment of data.



The controller completes when all items of the ROM are hashed. It will raise the finished flag and then wait until reset to start again. Upon reset the controller will enter the Get state and begin operation. Internally it runs a counter and the finished condition is true when the counter is all 1s.



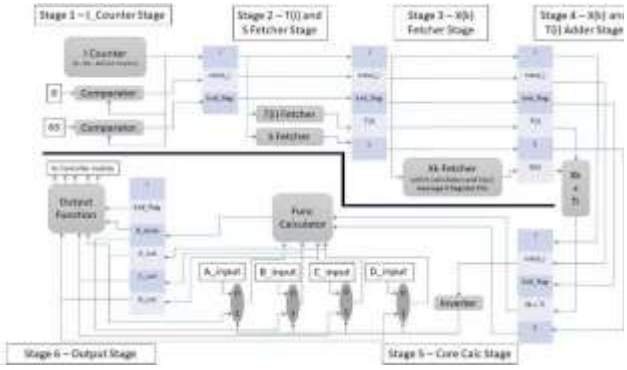
MD5 Core Module:

The MD5-Core Module is a 6-stage hardware accelerator that is focused on improving the hash generation per round of 512-bit block data. It is focused on the optimization and the calculation of the following equation

Function_output = B +
((A+Func(B,C,D)+X[k]+T[i])<<< s)

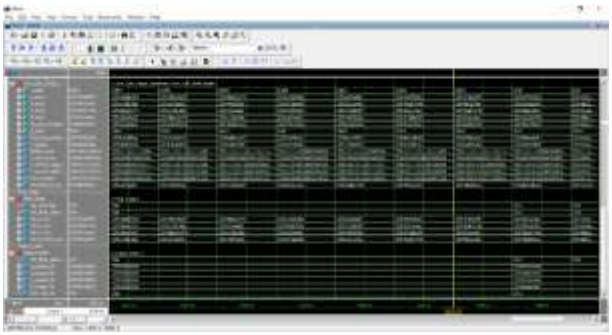
In the above equation, based on the value of the current step i.e., the variable i, the values of k, T[i] and s vary and can hence be tabulated. This tabulation is stored in the ROM as it remains the same over multiple rounds. Since these are memory access operations over a 64-bit quantity, there can be substantial amount of time delay in fetching the items and may require a dedicated clock for the same purposes. Therefore, the memory fetches and the X[k] + T[i] in the above equation is calculated

beforehand in stages before reaching the main calculation stage where the F, G, H or I function calculation happens and henceforth the remainder of the equation.



The above diagram is the data-path for the 6-stage MD5 Core Module. Prior to all the stage calculations and iteration of I over multiple rounds, the Core module will first receive, 16 – 32-bit words from the controller and will populate its Register File of 16x32 which is the working register file. This part happens before all calculations begin, thus not having to populate and de-populate the Register file on every round. The first stage is the I-Counter stage which counts from 0 to 63. There are 2 comparators, one for initial I detection at i=0 and the end I detection at i=63. These inferences are useful and are carried out throughout the stages. As there is considerable amount of timing slack available in this stage these comparisons are performed in advance. The second stage is the T[i] and S fetcher stage. As these two items are variable over 64 rounds of I, they will require substantial memory access time and are given a dedicated stage. Therefore, these memory accesses are done in parallel and the inferences are carried to the next round. In the next stage, i.e.., the X(k) fetcher stage, there are two sequential memory access operations. Firstly, the value of k is fetched based on the current value of I and this value of k is used to fetch the 32-bit item from the input message of X(k) which is organized as 16x32 register file. In the next stage, is the Xk and Ti addition stage. Although this is a relatively light stage with only the addition to do, this stage will probably relieve some load from the next stage which is the Core Calc stage where there is numerous addition, shifting and logical operations are being done. Further this addition cannot be shifted to the earlier stages also as they are memory access stages and may require long time slacks and may not accommodate an adder after the memory access. The next stage is the Core Calc stage where most of the calculation is done and can easily be inferred to consume the maximum time slack. In this stage the following events occur – The value of A, B, C and D buffers are chosen depending on the value of I – if it is required to fetch the initial values of A, B, C, D or from the forwarding multiplexers from stage 6. Once the values of A, B, C and D are chosen, they are fed into the Func_Calc module and simultaneously, they are written into the stage 5 / stage 6 registers in the order A ← D; B ← A; C ← B; D ← C thus satisfying equation [2]. The Func Calc module first performs X (B, C, D) based on the value of I, where X = {F, G, H, I}. In the next phase, the calculation of F + A + (Xk_plus_Ti) is performed and then the circular shift by s is performed. As the last calculation, it is added with the B contents and provides the result. This result is then fed into the B register of the stage 5 / stage 6 register. This stage is purposefully made into a single stage to accommodate all the calculations as the forwarding unit from stage6 is relatively simpler. Further simplification of the stages can also be approached, however as referenced by the base paper, it may not provide the necessary tradeoff in performance vs area. Hence this six-stage approach is used. In the last stage i.e., the output stage, there is not much calculation except that when i=63 in this stage, the End of conversion signal is raised and the calculation X_out = X_current + X_initial is performed where X = {A, B, C, D}. Hence the MD5 Core module can perform one round of calculation in 70 clock cycles which is broken down as - 7 cycles for the 1st step output to be calculated + 63 cycles for remaining 63 steps. The figure is in comparable numbers with the base material as cited in [4] and much more efficient compared to software implementation which require nearly 1600 clock cycles. The pseudocode followed is the same one as mentioned in [5].

The implementation was done in Verilog and a sample test of the module consisting of the input text “Hello World” and its output is provided below.



Below is an image of the controller running all the way through.

In the image we can see that the complete flag is raised at time 1493.6 microseconds. The amount of data that we process in this simulation is a 1024 location array of data. This allows us to find that we are taking 1.46 microseconds per data unit. When converting this to clocks using the 20 ns clock that we used, we can find that we used 73 clocks per data unit. From 1600 clocks, we can say that we have made about a 22x improvement over the software design.

IV. Challenges

The challenges faced during the implementation of the project were manifold. In addition to the above, it is to be noted that there are also certain restrictions and special pointers to be taken into consideration. All data to be processed from the input message data to the buffer contents are in little endian format. The shift operation used here is the circular shift, which is not readily available in Verilog, which has only arithmetic and logical shifts. Rather than going for a shift register type of operation which achieves the circular shift in multiple clocks, certain mathematical procedures along with the available arithmetic shift was used to achieve the circular shift in one cycle. Additionally, all the memory accessing elements such as the T(i), X(k) and S elements were required to provide the outputs at that very clock so that they can be propagated through the stages. The architectural design, however produced a significant challenge as a novel architecture with the controller and core concept had to be designed with their interfaces designed and defined. This had to be efficient, parallel and scalable to process any chunk of 512-bit data. The current design can still be optimized, by having two register set files and making the controller module to populate the one set of register file in anticipation of the next round while the core module works with the other. In this way, the idle time when the core is busy, can be used by the controller to populate the Register file data for the next round instead of having to wait till the end of the round, to populate the only register file and passing the pointer to the register file as an argument to the core. This apparent advantage in speed comes with an increase in another set of 16x32 register file. However, citing limited resources, the current approach was adopted and can be modified to meet the proposed design in the future.

V. Novelty

Our approach for acceleration of the MD5 algorithm utilized both pipelining and FPGA-based hardware acceleration. The MD5 core logic was divided into six stages to improve hash chain generation. However, further decomposing introduced the tradeoff in performance and area. Hardware acceleration has proven to significantly speed up computation of the computationally intensive MD5 algorithm. Whereas pipelining aims at speeding up each round of the MD5 algorithm, our hardware implementation focused on two ways of improving the speed of the hash generation: within an individual round or over multiple rounds. Our application space, hash chaining, is a newer one and therefore is not saturated by many papers on the subject. We decided to take an approach we had not yet seen to the problem by introducing pipelining and hardware acceleration to speed up the generation of a hash chain. By utilizing a combination of pipelining and hardware acceleration, our implementation was able to significantly improve the performance of the MD5 algorithm compared to the software implementation.

VI. Conclusion

The hardware implementation that we created was able to significantly improve over the software design. Using the design that we created, data verification for large chains can be sped up immensely to be real time. This will help minimize delay as security becomes more and more important to applications and things like long chain authentication become a necessity to

everyday tasks like logins. We hope that our work can be used to increase the adoption of long chain data verification techniques and increase general system security for applications.

VII. CITATIONS

1. A. Cilardo and N. Mazzocca, "Exploiting Vulnerabilities in Cryptographic Hash Functions Based on Reconfigurable Hardware," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 5, pp. 810–820, 2013.

2. A. Thiruneelakandan and T. Thirumurugan, "An approach towards improved cyber security by hardware acceleration of OpenSSL cryptographic functions," 2011 *International Conference on Electronics, Communication and Computing Technologies*, 2011.

3. K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware Implementation Analysis of the MD5 Hash Algorithm," *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, Jun. 2005.

4. Anh Tuan Hoang, Katsuhiro Yamazaki and Shigeru Oyanagi "Multi-stage Pipelining MD5 Implementations on
FPGA with Data Forwarding", *16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008.

5. "MD5," Wikipedia, 27-Nov-2018. [Online]. Available: https://en.wikipedia.org/wiki/MD5. [Accessed: 01-Dec-2018].