MD5 Hardware Accelerator

Team Members:
Harrison Getter
Travis Dang
Ebrahim Alhaddad

5th December 2018

1. Abstract

With the popularization of cryptocurrency, blockchain technology has been found desirable in other applications. In blockchain technology ledgers or transactions are assigned hash values that are computationally intensive and time consuming to complete. ASIC which process hash functions efficiently and quickly are in more demand than ever before. In this paper, we are outlining our approach to fulfill the need for hash function ASIC. Message Digest 5 Algorithm (MD5) is a hash function which generates a 128-bit value. MD5 is widely used in cryptography and would benefit from dedicated hardware. First, we would implement the algorithm in software (C++). We implemented two hardware accelerators first by using a single core processor and second by pipelining it through a network on chip. Finally we analyzed the power, area and timing of the respective implementations that

2. Introduction and Motivation

Cryptocurrency has been a widely popular and publicized topic within the past year. As a result, blockchain has seen a rise in applications. Blockchain is a process of recording, verifying, and approving transactions or ledgers with a decentralized network. The industry most interested in blockchain is the financial industry where transparency and transaction verification are necessary. Supply chain management, data sharing, medical record keeping, and internet of things networks are areas of interest for the technology [1].

To verify transactions in blockchain, hash values are assigned to each transaction block and must be solved by computers in the network [2]. Hash functions are computationally intensive and must be run a significant number of times. Hashes are the main source of processor use and power consumption in blockchain technology. As the demand for blockchain technology grows, the need for hash dedicated hardware is evident. We seek to solve this problem by implementing MD5 hash function as a SoC.

By creating a SoC dedicated to solving hash functions, we can reduce the power consumption to processes hashes, as well as increase throughput [3]. We chose to design our SoC using an NoC design because it best suited our pipelined approach to accelerating our selected algorithm which is MD5.

3. Previous Work

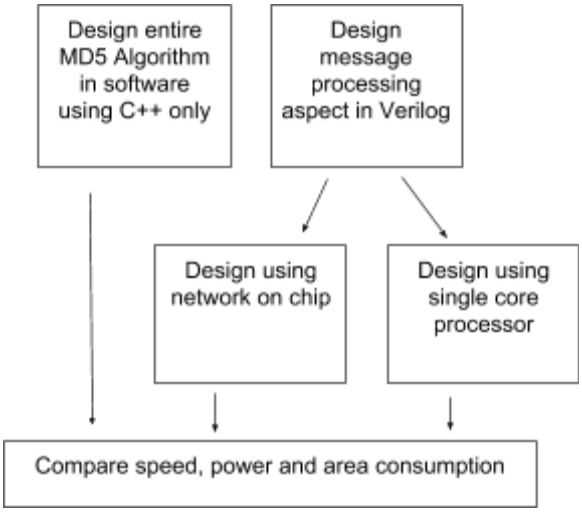
There are two previous works that we have researched that led us to our project. The first is a paper by Ronald L. Rivest from MIT Lab for Computer Science called The MD5 Message-Digest Algorithm. In this paper, Rivest describes the Message Digest (MD5) algorithm that we have accelerated. The MD5 algorithm consists of four main stages: Appending padding bits; Appending length; Buffer initialization; and Message processing [2]. If the message being encoded is not a multiple of 64-bytes, then the algorithm appends 0's until the size of the message is a multiple of 64-bytes. The next step is appending the length. In this step, it appends a 64-bit representation of the message to the end of the message created in the previous step. After this, it initializes 4 32-bit constants used in the algorithm. The fourth step is the most important step. This is where it computes the 16 byte encoded hash. First it breaks the 64-byte block into 16 4-byte chunks and performs a series of 4 bit manipulation iterations defined in [2] on each of the 16 chunks. This process is repeated until all blocks are encoded. Once they are encoded, they dissolve end up dissolving into a singular 16 byte sequence.

The second paper we looked into inspired our idea to pipeline the MD5 algorithm. K. Jarvinen, M. Tommiska and J. Skytta's paper named Hardware Implementation Analysis of the MD5 Hash Algorithm examines the effects of pipelining on delay, area requirements and throughput is performed. Their goal was to improve speed and throughput. To do this they decided to implement the MD5 algorithm on an FPGA. They implemented Rivest

fourth step in a pipeline hardware architecture and keep the first three steps in software. After implementing the pipelining, they created multiple instances of the pipeline to increase speed. Their final design had 64 stages which demonstrated the fastest speed and throughput, but also used the most area and power.

4. Approach

4.1 FlowChart



4.2 Software

The purpose of the software implementation is to provide a reference to validate the hardware designs' functionality and to compare their performances. The source code in C++ is imported from the hashlib++ library [4], which is similar to the C code presented in the MD5 algorithm paper used to build the hardware design [5]. This approach allowed the C++ implementation to perform as proposed in the paper so it can be used with confidence to validate our other designs.

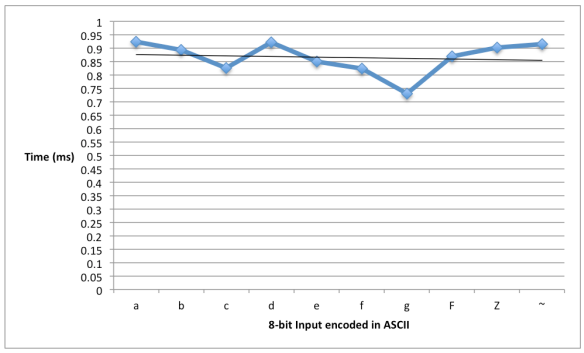
A few modifications were necessary to include timing analysis and provide a framework for debugging the hardware design. Given the algorithm consists of four computational stages, the code invoked these stages in four separate functions sequentially while updating the states A, B, C and D internally.

```
md5Round1(current_state){
//call bit-manipulation function F and modify the input parameter
}
md5Round2(current_state){
//call bit-manipulation function G and modify the input parameter
}
……..
```

Input padding was removed given it is not objectively analyzed in this project. The hardware design imposed a limitation on input given the FPGA could not accept 512-bit input signals. A workaround was to provide an 8-bit signal and use it construct a repetitive sequence 512-bit long. Similarly, the input preparation sequence of the code simulated the aforementioned process in order to run the different designs using identical inputs.

```
Unsigned char input = ….
Unsigned char* md5Message = new char[64]
for(int i = 0; i < 64; i++){
        md5Message[i] = input
}
```

After receiving the 8-bit input value, encoded as a single char type, the code will generate the hash and print out all the states' values after each stage. By comparing these results with that of the hardware design, debugging became more efficient. Moreover, the code uses the standard chrono library to measure computation time at a nanosecond resolution. The timing function only targeted the computational functions and did not measure the time spent on processes irrelevant to analysis like printing functions. The program calls to functions processing the same input message. One function prints out the state values while the other only invokes the md5 stages with timing.



The graph presents the code performance given a set of 8-bit input encoded as characters. The graph shows that the the software implementation performs at an average of 875ns. Different inputs resulted in small variations fluctuating between a maximum of 920ns and a minimum of 725ns. These fluctuations can be attributed to external processes on the machine. It is difficult to analyze further the significance of these processes given the program cannot take advantage of the machine's entire processing power, which is an inherent drawback in the software implementation. For comparative analysis purposes, it is assumed that the software performance is stable at the given average of 875ns, which is used as a reference point to compare the time-efficiency of the hardware implementation.

4.4 Hardware

4.4.1 Single Core

We implemented the MD5 processing algorithm in Verilog. The input was an 8-bit input and it was repeated to 512-bits. It was split into 16 submessages. We created verilog functions to perform the repetitive auxiliary functions and main operations. We called those functions to update A,B,C,D values for each stage. The operations were all performed in one always block.

4.4.1.1 Single Core and Software Pseudo Code

The program receives an 8 bit input from the user. (message_in)
The input is repeated for 512 bits and stored as a message. (message)
The message is broken down into 16 submessages. (M0-M15)
A, B, C, D are initialized into 0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210 respectively.
A, B, C, D are processed using the following formula:

w = x + (w + fn(x, y, z) + Mj + T[i] <<< s)

where fn can be one of four auxillary functions that perform bitwise operations,
Mj is a submessage, T[i] is 4294967296 * abs(sin(i)), and s is rotate amount.
A, B, C, D can be substituted for w, x, y, z determined by the algorithm

At the end of the function, the output is A, B, C, D and these can be concatenated to become the hash output.
The source code is located in the PE.v.

4.4.2 Network on Chip

As explained in the [3], the algorithm has computationally intensive section where it preprocesses the message and creates the output hash. We decided to implement this section in hardware because it takes the most time to compute. To do this, we used the network on chip model provided in lab 6 but used a 2x2 mesh. The MD5 algorithm has 4 main functions, F (X,Y,Z), G (X,Y,Z), H (X,Y,Z), I (X,Y,Z), and can be found in the NoC.v file. We dedicated each node to a different function so each processor was uniquely designed. Next, we pipelined the algorithm through all of the nodes to generate the correct hash.

4.4.2.1 Network on Chip Pseudo Code

The top module receives an input from the use. (message_in)
The top module links the data to the first processing element. (message_in -> R1)
R1 Module
The message_in is repeated for 512 bits and stored as a message. (message)
The message is broken down into 16 submessages. (M0-M15)
A, B, C, D are initialized into 0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210 respectively.
A, B, C, D are processed using the following formula:

w = x + (w + fn(x, y, z) + Mj + T[i] <<< s)
where fn is one of four auxiliary functions that perform bitwise operations,
Mj is a submessage, T[i] is 4294967296 * abs(sin(i)), and s is rotate amount.
A, B, C, D can be substituted for w, x, y, z determined by the algorithm

The first stage has the operation performed 16 times with different inputs determined by the algorithm.
A modified version of A, B, C, D is sent as output in two clocks.
The top module links the output data from the processing element and links it to the interface.
The output flit is linked to the network on chip.
The flit is received by the destination processing element.
The second stage has the operation with a different auxiliary function performed 16 times with different inputs by the algorithm.
A modified version of A, B, C, D is sent as output in two clocks.
The top module links the output data from the processing element and links it to the interface.
The output flit is linked to the network on chip.
The flit is received by the destination processing element.
The process is repeated for a third and fourth stage.
At the end of the function, the output is A, B, C, D and these can be concatenated to become the hash output.
The source code is located in files top.v, interface.v, and PE.v.

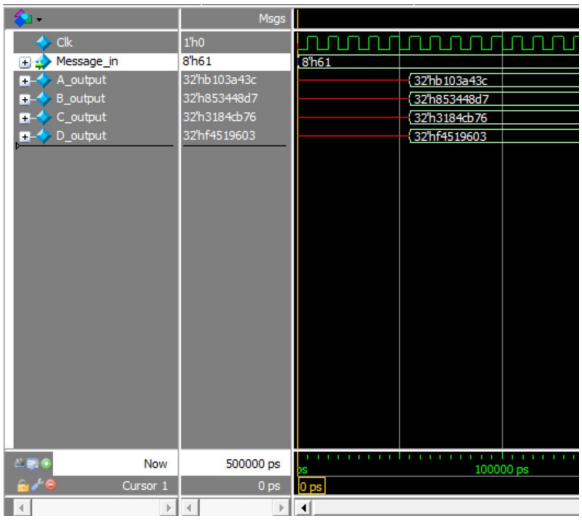
4.4.3 Verification

Software Results

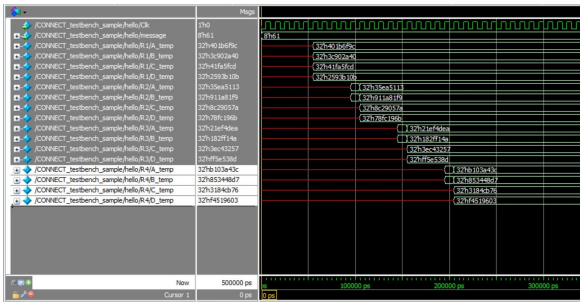
```
#include <iostream>
#include "md5.h"
#include <chrono>

int main(int argc, const char * argv[]) {

    //Create Input
    unsigned inputChar = 'a';
```

```
states after round1:
state[0] = 401b6f9c
state[1] = 3c902a40
state[2] = 41fa5fcd
state[3] = 2593b10b
*************
states after round2:
state[0] = 35ea5113
state[1] = 911a81f9
state[2] = 8c29057a
state[3] = 78fc196b
*************
states after round3:
state[0] = 21ef4dea
state[1] = 182ff14a
state[2] = 3ec43257
state[3] = ff5e538d
*************
states after round4:
state[0] = b103a43c
state[1] = 853448d7
state[2] = 3184cb76
state[3] = f4519603
time taken: 858ns
Program ended with exit code: 0
```

Single Core Processor



NoC Results



4.5 Results

Below are the results from our implementations. Software is not included for area and power comparisons but is included for timing. Hardware is included for all three area, power consumption and timing.

4.5.1 Timing Comparison

	Time (ns)
Software Implementation	875
Hardware Implementation (Dedicated)	10
Hardware Implementation (NoC)	220

Single Processor Clock Speed

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	10.000	100.000

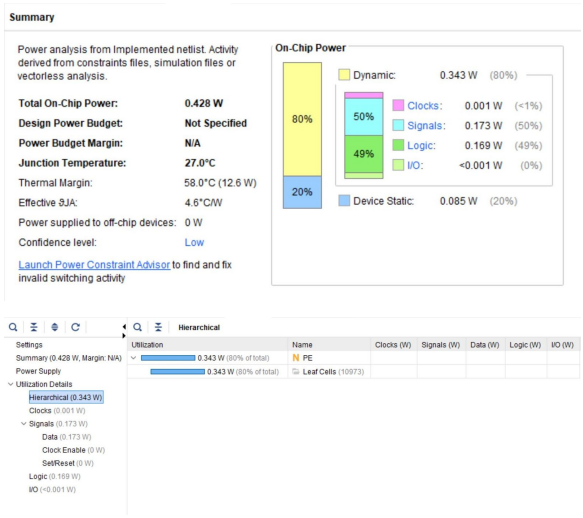
NoC Clock Speed

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	110.000	9.091

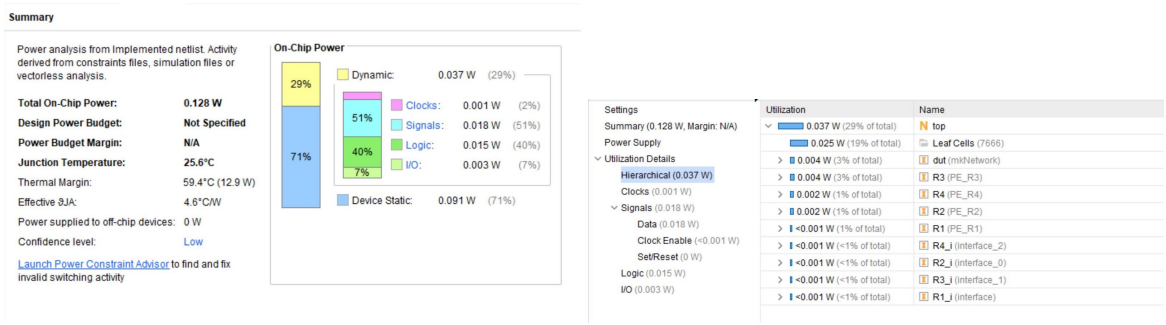
Result: For the dedicated processor, each clock period outputs a complete hash. For the NoC implementation, it takes two clocks to output the result because of the NoC flit size limitation. As a result, the dedicated processor is 20x faster than the NoC implementation.

4.5.2 Power Utilization

Single Processor Power Summary



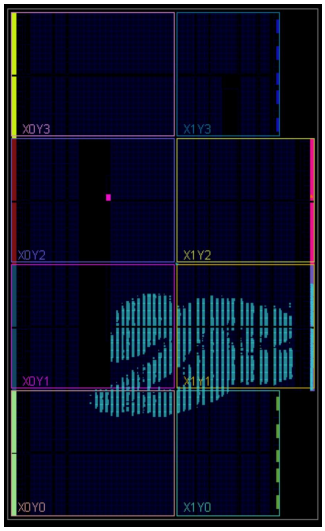
NoC Power Utilization



Result: The NoC implementation has significantly less power consumption than the dedicated hash processor. The NoC requires only ⅓ of the power the dedicated hash processor requires.

4.5.2 Area Utilization

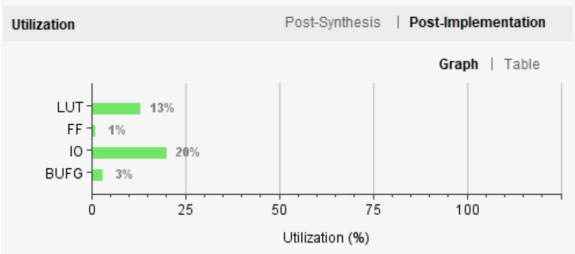
Single Processor Area Utilization



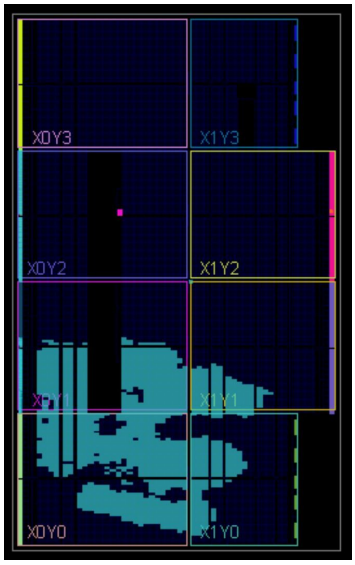
Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	7997	63400	12.61
FF	34	126800	0.03
IO	43	210	20.48
BUFG	1	32	3.13



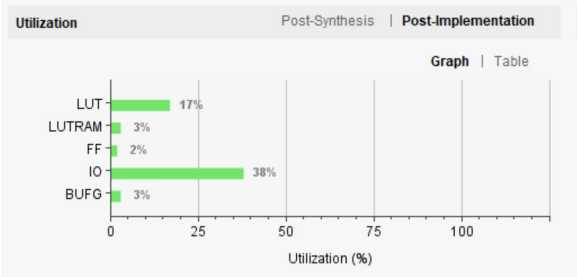
NoC Area Utilization



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	10860	63400	17.13
LUTRAM	624	19000	3.28
FF	2042	126800	1.61
IO	79	210	37.62
BUFG	1	32	3.13



Result: The area utilization is slightly larger for the NoC implementation, however, it is not by a large margin. The differences is a 4% increase in LUT utilization and an 18% IO utilization. This large increase is most likely due to how we implemented the NoC. In the NoC, the input is sent to each processing element in order to recreate the message. This would increase the IO utilization and we can reduce this by sending a copy of the input to each processing element directly instead of sending the IO pin directly.

5. Challenges

The biggest constraint on our project was the frequency limitations of the network on chip module. According to the designers, the network on chip module has frequency issues due to the use of LUTs which require long interconnect wires [6]. This proved to be true with our results. The hash dedicated processor was able to have a 100 MHz clock while the network on chip required a 9 MHz clock despite having smaller blocks on logic for each processing element.

The next challenge was being able to accurately depict the benefits of a network on chip. The benefits of network on chip are seen in high traffic situations. It is difficult to simulate this because the current network on chip took up a large area and making the network on chip larger could cause us to fail implementation. It was also difficult to simulate traffic. As a result, the full benefit of the network on chip has yet to be utilized.

The next issues are minor. We implemented a 64-bit data network on chip which is not large enough to send all data in one cycle. Therefore, we had to send data in two cycles which reduces the hash throughput in half. There were not enough inputs for a 512-bit message, so we have an 8 bit input that is repeated until it reaches 512 bits and is input as the message.

6. Conclusion

We learned from our results that implementing the MD5 algorithm is faster using any hardware, but all hardware comes with a cost. On the hardware side, we learned that using a single core pipeline SoC is faster and requires less area than our NoC pipeline design, but dramatically increases the power consumption.

We also learned that we did not chose the most optimal number of stages for our pipeline. K. Jarvinen, M. Tommiska, and J. Skytta paper named Hardware Implementation Analysis of the MD5 Hash Algorithm compares the gate count, frequency and speed of each stage they analyzed. After receiving our phase 2 feedback to explain how we decided upon our number of stages, we realized that 16 stages would have been more efficient. The first four columns for the table below shows the constants described in K. Jarvinen, M. Tommiska, and J. Skytta's paper. The next three columns provide better metrics to compare the stages because they provide a more holistic approach by comparing multiple variables at the same time. In the end, it is clear that a 16 stage pipeline would be the most optimal throughput because there is a significant increase in Area x Power and Area / Speed from stage 16 to stage 32.

Stages	Area (Gate Count)	Power per Area (Frequency in MHz)	Speed (Throughput in Mbps)	Area x Power	Area x Power Difference	Area/Speed	Area Speed Difference	(Area x Power)/Speed
2	31200	78.3	607	2442960		51.40032949		4024.645799
4	43051	80.7	626	3474215.7	1031255.7	68.7715655	17.37123601	5549.865335
8	61727	80.7	626	4981368.9	1507153.2	98.60543131	29.83386581	7957.458307
16	81191	80.7	626	6552113.7	1570744.8	129.6980831	31.09265176	10466.6353
32	118803	84.1	652	9991332.3	3439218.6	182.2131902	52.51510712	15324.12929
64	169715	93.4	725	15851381	5860048.7	234.0896552	51.87646499	21863.97379

[3]

7. References

[1] Underwood S. Blockchain Beyond Bitcoin. Communications of the ACM. 2016;59(11):15-17. doi:10.1145/2994581.

[2] Nofer, M., Gomber, P., Hinz, O. et al. Bus Inf Syst Eng (2017) 59: 183. https://doi-org.libproxy2.usc.edu/10.1007/s12599-017-0467-3

[3] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware Implementation Analysis of the MD5 Hash Algorithm," IEEE Xplore Digital Library, 24-Jan-2005. [Online]. Available: https://ieeexplore-ieee-org.libproxy2.usc.edu/document/1385853. [Accessed: 28-Sep-2018].

[4] B. G. Delbach, "hashlib ," Hashlib2plus. Sourceforge, 13-Oct-2011.

[5] R. L. Rivest, " The MD5 Message-Digest Algorithm," *Network Working Group, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, Apr. 1992.

[6] Papamichael, Michael & C. Hoe, James. (2012). CONNECT: Re-examining conventional wisdom for designing nocs in the context of FPGAs. 37-46. 10.1145/2145694.2145703.