Mohammed Tanner

2/16/2023

SHA-256 Hashing Hardware vs Software Implementation

Abstract:

This personal project aims to implement the SHA256 hashing algorithm using Python and a Field-Programmable Gate Array (FPGA). The project will begin by studying hashing in-depth to understand the mathematical principles behind it.

Next, the project will use Python to create the initial implementation of the SHA256 hashing algorithm. The implementation will be tested using various test cases to ensure its accuracy and performance.

In the second phase, the project will use an FPGA to implement the hashing algorithm. The FPGA implementation will be optimized to improve the speed and efficiency of the encryption and hashing process. Design space exploration will be done to find the best design trade off between speed and space used by the hardware.

Finally, the Python implementation and the FPGA implementation will be compared and evaluated based on their performance metrics. This project will help in developing a better understanding of hashing algorithms and hardware implementation, as well as provide hands-on experience with FPGA design and implementation.

What is SHA-256:

SHA-256 is a cryptographic hashing algorithm that takes input data and produces a fixed-length output(64 hex characters which is 256 bits), called a hash value or digest. The algorithm is designed to be one-way, different from encryption. In other words, you cannot decrypt the original message from a hash[1]. You can only reproduce a hash by knowing the original input. This makes SHA256 widely used in applications such as digital signatures, password storage, and secure communication.

Inputs and Outputs:

The input for this project will be a message or data that needs to be hashed. The output will be the hash, a fixed-length string of characters that represents the input data. The hash output is typically used for verifying the integrity of passwords.

An example of possible input and it’s output is:
INPUT = “abc”
OUTPUT = ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

INPUT = “apple”
OUTPUT = 3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b

Processing:

For SHA256 hashing, the following processing steps are performed on the input message:

- Message Padding: The input message is padded to a multiple of 512 bits.
- Message Parsing: The padded message is divided into 512-bit blocks.
- Message Schedule: The block is processed through a message schedule to produce a set of intermediate values.
- Compression Function: The intermediate values are then processed through a compression function, which combines the values to produce the final hash output.

Code:

The code used is from an online implementation found on github by user keanmind. The implementation uses FIFO which doesn’t work for our purposes. The implementation was changed to use AXI Streaming. sha256.cpp code performs the steps mentioned in the procession section. Please note, SHA-256 yields different results based on implementation and number of rounds so it is natural to have different results than the online tool(CyberChef) we will use to compare some of the outputs. The following is a quick walk through of this code:

1. The function sha256() is defined with one input stream and one output stream (idata and ohash) for the input data and output hash respectively.
2. The function is also marked with pragmas (#pragma) to indicate the interfaces for the input and output streams and the return type of the function.

3. Inside the function, temporary variables (tmp8, stateREG, data, and hash) are defined. The variable tmp8 is used to perform the streaming of axi_stream data into the local “data” variable.
4. The load_data loop reads in the input data (idata) from the stream and stores it in the data array.
5. The convert_to_words loop converts the 64 bytes of input data in the data array into 16 WORDs (32-bit integers) and stores them in the m array.
6. The create_schedule loop generates the remaining 48 WORDs in the m array using the SHA-256 schedule algorithm.
7. The initial values for the SHA-256 hash algorithm are loaded into a, b, c, d, e, f, g, and h variables.
8. The compression loop then performs the main compression function of the SHA-256 algorithm, updating the values of a, b, c, d, e, f, g, and h for each of the 64 rounds.
9. The final values of a, b, c, d, e, f, g, and h are added to the initial values in the stateREG array.
10. The convert_endianess loop converts the stateREG array into a 256-bit hash in little endian byte order, storing it in the hash array.
11. The store_hash loop finally writes the 32 bytes of the hash array into the output stream (ohash).

Evaluation:

A valid SHA-256 hash must be 64 hex characters long which is 256 bits. In other words, as long as our hash is reproducible with the same input and is 64 chars long, then we have a valid and functional hashing algorithm. To test the functionality of this code, a test bench was written(shatest.cpp). The correct output for the hashing algorithm can be compared to the output using CyberChef, an online tool which encrypts, decrypts, and hashes. The test bench inputs “abc” as an array of hex values, each translating to a character using ASCII. The code is verified using the following steps:

1. Declare two data streams: idata, and ohash. These streams are used to communicate between different blocks of the HLS design.
2. Define an input array of unsigned characters called "input," which is initialized with a specific set of values.

3. Iterate through the "input" array and for each element, create a new ap_axiu data structure (with data width of 8 bits, 2 bits for user data, 5 bits for ID, and 6 bits for destination tag), assign the current element of the array to the "data" field of the ap_axiu structure, and write this structure to the idata stream.
4. Call the sha256 function with the idata and ohash streams as inputs.
5. Define an output array of unsigned characters called "output."
6. Iterate through the output array and for each element, read a data structure from the ohash stream, assign the "data" field of this structure to the current element of the output array, and then discard the data structure.
7. Print out the values of the output array in hexadecimal format to the console.
8. Return 0 to indicate successful completion of the program.

In addition to the test bench in vitis, the functionality of the hardware is verified through multiple test cases in the Jupyter notebooks submitted with the project.

Hardware vs Software:

For the purposes of this project, the entirety of SHA256 was implemented in hardware instead of splitting up the design into hardware and software components. Design space exploration is performed to compare different possible designs using HLS Vitis and the different directives it provides. The best design is to be then compared with a software implementation to demonstrate the strengths of hardware compared to software.

Hashing is much faster in hardware compared to software due to the parallelism and dedicated hardware resources that are available in hardware. In hardware, a single operation can be performed on multiple data streams simultaneously, increasing the throughput of the operation. Additionally, dedicated hardware resources such as multipliers and adders can be utilized in hardware to perform hashing more efficiently.

In software, hash computations are typically performed using iterative algorithms, which require multiple iterations to compute the hash. This process is typically done sequentially, which means that each iteration must wait for the previous one to complete, resulting in slower execution times. Additionally, the hardware resources available in software are typically shared among various tasks, which can result in slower hash computation times due to contention for those resources.

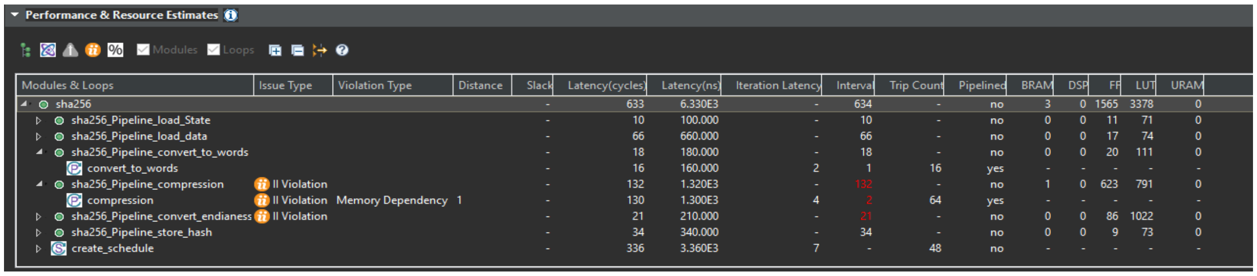
The following is a table showing average execution time of software and hardware after a number of example runs,

Average Software Execution Time(seconds)	Average Hardware Execution Time(seconds)
0.008626	0.003542

In summary, due to the increase in throughput using pipelining and unrolling and the mathematical operations of SHA256. Hardware is clearly superior when it comes to cryptographic algorithms.

<u>Design Space Exploration:</u>

After synthesis of the initial code, data dependencies and violations were shown as can be seen below,



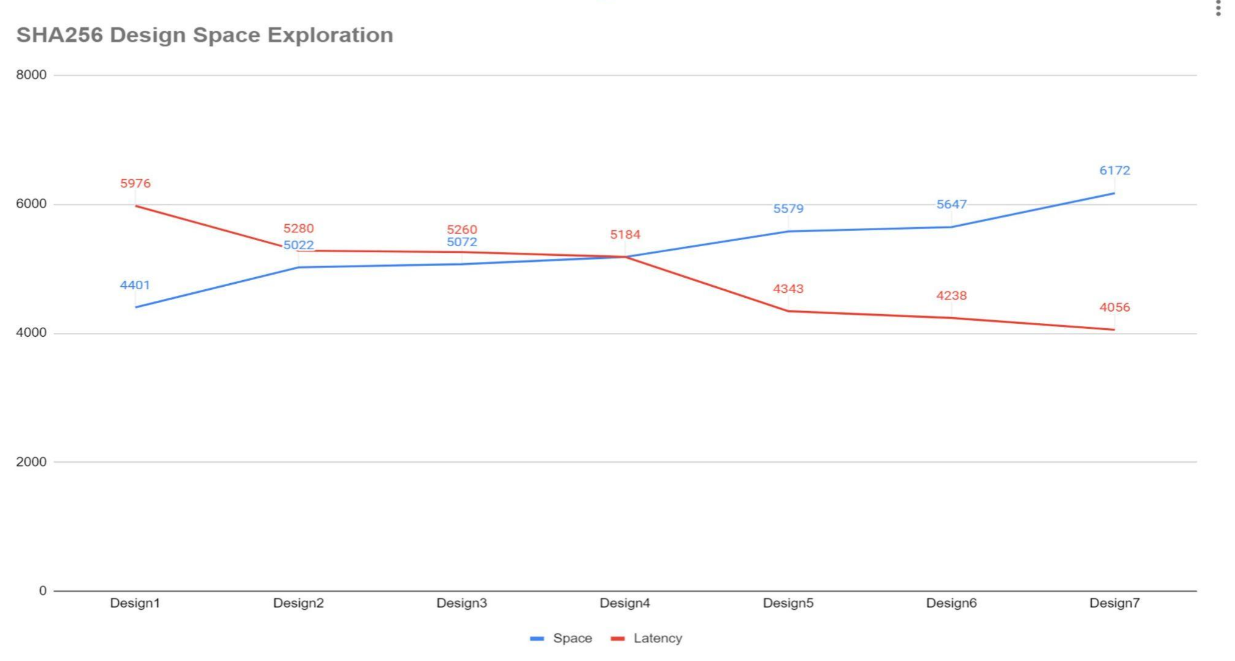
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
sha256				-	633	6.330E3	-	634	-	no	3	0	1565	3378	0
sha256_Pipeline_load_State				-	10	100.000	-	10	-	no	0	0	11	71	0
sha256_Pipeline_load_data				-	66	660.000	-	66	-	no	0	0	17	74	0
sha256_Pipeline_convert_to_words				-	18	180.000	-	18	-	no	0	0	20	111	0
convert_to_words				-	16	160.000	2	1	16	yes	-	-	-	-	-
sha256_Pipeline_compression	II Violation			-	132	1.320E3	-	132	-	no	1	0	623	791	0
compression	II Violation	Memory Dependency	1	-	130	1.300E3	4	2	64	yes	-	-	-	-	-
sha256_Pipeline_convert_endianess	II Violation			-	21	210.000	-	21	-	no	0	0	86	1022	0
sha256_Pipeline_store_hash				-	34	340.000	-	34	-	no	0	0	9	73	0
create_schedule				-	336	3.360E3	7	-	48	no	-	-	-	-	-

These issues were solved by increasing the clock period to 12ns. The following table shows optimization steps taken to find the best possible design. Other designs were recorded, however, this table strictly shows improvements where the percentages in green are increases of latency and those in red are increases in space required for the design. The clock cycle period was increased to 13ns on design 5 due to data dependency violation when pipelining the create_schedule loop.

Optimization	Description	Latency(ns)	Space(max(LUT,FF) + 100*DSP)
1	**BASE DESIGN(no pipelining)**	5976	4401

2	**Pipeline compression and convert_endian loop with II = 2, 4**	5280(11.65%)	5022(12.4%)
3	**Convert_endian pipelining = off**	5260(0.38%)	5072(0.99%)
4	**Compression and convert_endian II = 2, 2**	5184(1.44%)	5184(0.98%)
5	**Added create_schedule pipelining II = 1(CHANGED PERIOD TO 13NS)**	4342(16.24%)	5579(7.08%)
6	**Unroll convert_to_words by 2**	4238(2.40%)	5647(1.22%)
7	**Fully Unrolled load_data**	4056(4.29%)	6172(8.51%)
8	**Data Type optimization**	4056(0%)	6172(FFs DROPPED SIGNIFICANTLY)
	FINAL IMPROVEMENT	36.4%	31.18%

The following is a pareto graph showing latency and space of each design,



Design 4 seems like the most balanced tradeoff between latency and space. However, an approximate increase of only 1771 in space for a massive decrease of 1920ns in latency is quite impressive. While it depends on the specific scenario we are designing the algorithm for, the design with the lowest latency was chosen for our final design.

<u>Lessons Learned:</u>

1. Hardware debugging skills: debugging in hardware can be quite challenging. We got to learn how to check specific details of where the problem is occurring in synthesis. Such issues are pipeline timings and scheduling. Clock cycle period was an important part of making this design work. Using 10ns per cycle introduced scheduling issues, increasing it to 12ns was the shortest possible period. Also we learned that certain directives such as pipelining and unrolling can make the resulting output incorrect in certain scenarios.
2. Implementing AXI Stream can be a little challenging. We need a temporary variable of the same type of the stream to hold the current value to be passed into an array or for computation to be done on it.
3. Just because you can add/use a pragma doesn't mean it will help your design

<u>Next Steps:</u>

I would like to learn how to make an implementation which uses two input streams and one output stream. The additional input stream will be the preloaded states into the algorithm. These can be whatever the user wants instead of having fixed buffer states. I tried implementing an additional input stream, however I found it too challenging for me to understand it and have it implemented by the deadlines.

REFERENCES:

B. K. Jena, “What is SHA-256 algorithm: How it works and applications [2022 edition]: Simplilearn,” Simplilearn.com, 23-Feb-2023. [Online]. Available: https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm. [Accessed: 26-Apr-2023].