Literature Survey of SHA-256 Hardware implementations for ECSE 682 project

Shabbir Hussain
Dept. of Computer Engieering, McGill University
Montreal, Quebec
Email: shabbir.hussain@mail.mcgill.ca

***Abstract*—SHA-256 is a cryptographic one way hashing algorithm used to verify the authenticity of files and messages. It is a NIST standard used in many protocols such as SSH, PGP, IPSec and HMAC. This project compares a software implementation, versus a hardware implementation by a high level synthesis tool versus a fully hardware design implementation. The results show that in the hardware design implementation comes out ahead in terms of performance and area on a Cyclone V fpga.**

I. Introduction

Cryptography is often known as a practice to send secret messages. While encrypting and decrypting messages are an important part of keeping secrets, so too is the method of verifying that message content is unaltered from sender to receiver. One of the methods employed to make sure that a message remains unmodified in transit is called a Message Authentication Code (MAC). These codes are computed based on the message contents and a shared secret key between sender and receiver. A MAC provides message integrity because an imposter can not feasibly create a false message and MAC pair or modify the contents of an existing message that will trick the receiver into believing that the messages were authored by the true sender [1]. This is because the generation and verification of a MAC involves using a cryptographically secure hash function such as the SHA-256 algorithm. The SHA-256 algorithm is a NIST standard that is part of a family hash functions called the SHA2 family. These functions take as input an arbitrary length string of bits and compute fixed length string of bits called the Digest Message (DM). These functions are one way functions with a low chance of collisions. Changing 1 bit of the input dramatically changes the output making it very difficult to guess the input based on the output. This property is what gives it its relevance in the field of cryptography. The only way to compute the input a DM is to guess and as of writing this paper there no known attacks on the SHA-256 to speed up this process. Given a 256 bit DM it would take a modern computer several years to find the corresponding input. This is because cryptographic functions are slow in general.

Computing a MAC for every message adds a performance and power penalty. IPSecs performance bottleneck is the HMAC mechanism which is responsible for authenticating the transmitted data [2]. VLSI is literature is rich in solutions that offload the has computation to hardware to quickly compute hashes with a low area foot print.

This paper presents three implementations of the SHA-256 algorithm. The first approach is a fully software implementation run on a 12 core,Intel(R) Xeon(R) @ 2.50GHz computer located at legup.ece.mcgill.ca. The second approach sinthesises several implementations using the Legup High Level Synthesis (HLS) tool. The third approach implements the algorithm directly into an FPGA. This project compares all thee approaches on the metric of throughput measured in bits per second (bps). Also it compares the last two approaches in on the metric of area on a an fpga in terms of logic elements used. The Objective of this project is two fold. First and foremost, it is to learn about the advantages and disadvantage of using an HLS tool as well as learning application specific VLSI design techniques. Second, it is to measure the performance gains over an HLS tool when designing a hardware implementation of a specific algorithm.

II. SHA-256 algorithm

The SHA-256 algorithm takes in a bit string of a maximum size of 2^{64} bits and outputs a 256-bit output called the Digest Message (DM) [3]. The SHA-256 algorithm can be divided into three steps.

1) Padding
2) Data Initialisation
3) Round Hashing

During the Padding step, the input message is subdivided into 512 bit blocks. The last block is padded to a multiple of 512 bits. At the end of the contents of the last block a 1 bit is placed followed by zeroes. The last 64 bits of that block are used to store the message length. Figure 1 show an example of a padded input.

The next step is the data Initialisation. In this step many variables are initialized to values specified by the National Institute of Standards and Technology [3]. The DM takes its initial value, the constants take their values, and the message block is loaded into a vector called the message schedule. The initialization of the message schedule is show in figure 2.

The round hashing step happens once for every message block. During this round, the initial digest message is broken into 8 32-bit components and manipulated by a set of rotations and combinational logic 64 times in an inner loop. This inner loop can be seen in Figure 3. The functions in the inner loop and data initialization (sigma, MAJ, etc.) are combinational logic functions defined by NIST [3]. Finally, the after all the

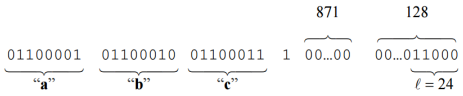

Fig. 1. An example of the SHA-256 padding

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

Fig. 2. The generation of the Message Schedule vector (W) based on the input Message (M)

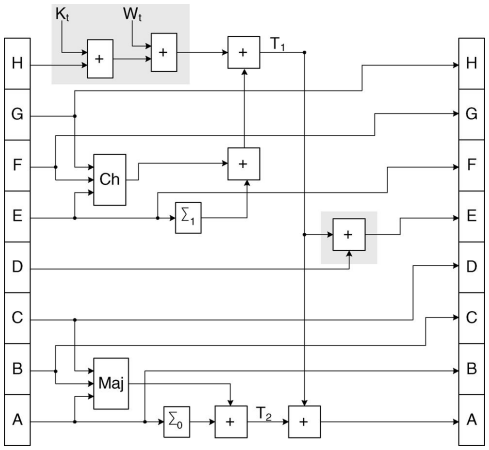

Fig. 3. One round of the inner loop of the SHA-256 algorithm

rounds are completed the digest message is recombined into a 512-bit format which is the result of the hash.

III. Software

The software implementation of the algorithm was written in C. Many C implementations exists already, such as the implementation in the OpenSSL library. However, a redesign from the ground up was the chosen route for two reasons. Firstly, it was for the educational benefit of the author. Secondly, this implementation needs to have the no amount of dependencies to libraries that are not compatible with the Legup HLS tool. This is so that the software implementation can serve as a building block for the legup HLS tool.

The C implementation followed the algorithm as described in [3]. In order to verify correctness, several inputs were compared to existing implementations. Some of the test cases are described in table I. The actual inputs are too large to display in this format and are omitted. These cases were chosen to test as many blocks of the code. For example, the no input case would test the inner loop of the program. If all operations are correct then the initial hash should be changed to ”e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495 991b7852b855”. An incorrect output could isolate an error at the inner loop block.

Once the code was completed it was measured for performance using a Gprof [4]. Gprof is a profiling tool for C code

TABLE I
Test cases to verify correctness of SHA-256 algorithm

Equivalence case	Passing
No Input	True
One block	True
Two blocks	True
All letters	True
256 letters	True

Fig. 4. Callgrind view of the program execution

used to measure execution time. Grof was able to accurately measure the execution time of the program to 4us. The program processed 512 bits. Using the formula $Throughput = bits/time$ we can calculate thoughput as $(512bits)/(4us) = 128Mbps$. These measurements were made on a relatively fast 12 core, Intel(R) Xeon(R) @ 2.50GHz computer. This computer is located at legeup.ece.mcgill.ca. This speed is impressive considering it is a fully sequential software implementation and reaches a throughput in the same order of magnitude to that of the implementation of [5] which had a throughput of 291 Mbps.

A tool call callgrind [6] was also used to profile the code. Callgrind outputs a nice visual representation of the execution time of each method of the program. This feature is useful for identifying hot spots in the program. Figure 4 shows the visual representation of the program execution. It is clear that the two most time consuming parts of the application are the padding (brown) and the round steps (red). This is expected because round steps are very sequential in the software implementation. We will use this information to speed up the hardware implementation in section V.

IV. Legup

Legup is a HLS tool which is able to take C code and compile it down to synthesisable verilog. The code from section III was reused in this section as the input to the Legup HLS tool. The setup requires the code to be located at legup.ece.mcgill.ca, a specialised make file and a specialised tcl file. The make file specifies the program, the location of the tcl and the location of other make targets. In this project the targets of the default Legup program were used by including the contents of Legup4.0/examples/Makefile.common. The tcl file specifies the FPGA board as well as sections of the code to parallelize. The hardware target for this design was the Cyclone IV DE2 board and also the Cyclone V GX board.

Several different configurations were tested to learn how a well an HLS tool can synthesize a given algorithm. The three legup designs are the full software implementation, the full hardware implementation and the openmp parallelized implementation. Unfortunately, the hybrid flow and loop pipelining flow were incompatible with this project. A full summary of all the implementations can be seen in table II.

A. Software flow implementation

Legup provides a software flow implementation. This flow compiles the C code for a MIPS processor. The MIPS processor runs at 50Mhz. When compiled with the make target $makesw$ the tigerMIPS processor is copied to working directory. Then to simulate the code the target $makeswsim$ was run. The resulting simulation ran in 4034 cycles. To calculate the throughput we can use $Throughput = bits * ClockRate/(cycles)$. In this case we obtain $512bits * 50Mhz/4034cycles = 6.32Mbps$. Which is very poor compared to previous software implementation. However, it is important to note that this processor is much slower.

B. Hardware flow implementation

Legup's default flow is the hardware flow. This flow takes the C code and compiles it down to synthesizable verilog, which we can then synthesize for our board. This whole flow is done with three targets $makehw$, $makep$, $makef$. The first target compiles the C code down to verilog, the second target creates a quartus project with the Cyclone IV DE-2 board as a target, the third target does does a full compilation for the target board. After this compilation process we can view the compilation reports to obtain the maximum clock frequency and the hardware utilization. We also run the target $makev$ to simulate the program in modelsim in order to obtain the number of clock cycles required to Hash 512 bits. From the reports we obtain an Fmax of 82.6Mhz. From the simulation we obtain a run time of 1677 cycles. We can then compute the Throughput to be $512 * 82.6/1677 = 25.21Mbps$.

C. OpenMP parallelization flow

This flow work the same as the previous, however, we change two pieces of the source. First in the source we add open mp pragmas to select pieces of the code where parallelism would improve performance. We parallelize the data initialization and the innerloop. Figure 5 and figure 6 show the open mp pragmas. Second, we add to a parameter to the tcl file to enable open mp parallelization. From this flow we obtained an Fmax of 84.87 Mhz and a simulation of 2237 cycles which resulted in a throughput of $512 * 84.87/2237 = 19.42Mbps$. Although it has a faster clock rate, it required more execution cycles to complete. This is most likely due to the overhead of using a large number of threads.

D. Hardware flow on Cyclone V

In order to verify the change in performance and area on a different generation of FPGA, the parallel hardware flow was recompiled for the Cyclone V. To do so, the tcl file

```
#pragma omp parallel num_threads (16) private (t)
for(t=0;t<16;t++){
    WORD w1 = (Mi[4*t]<<24) + (Mi[4*t+1]<<16);
    WORD w2 = (Mi[4*t+2]<<8) + Mi[4*t+3];
    W[t] = w1+w2;
    //printf("%04x\n",W[t]);
}
```

Fig. 5. Parallelizing the Data initialization

```
#pragma omp parallel num_threads (64) private (t)
for(t=0; t<64; t++){
    T1 = h+  EP1(e) + CH(e,f,g) + k[t] + W[t];
    T2 = EP0(a) + MAJ(a,b,c);
    h=g;
    g=f;
    f=e;
    e=d+T1;
    d=c;
    c=b;
    b=a;
    a= T1+T2;
}
```

Fig. 6. Parallelizing the inner loop

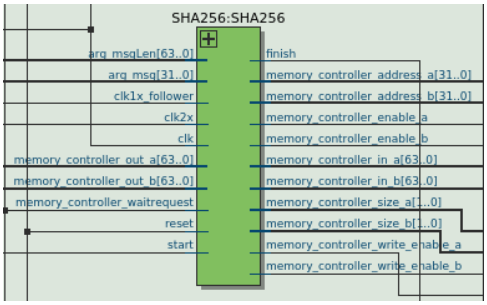

Fig. 7. The RTL view of the SHA-256 module

parameter for hardware target was change to the Cyclone V. From this flow we obtained an Fmax of 130 Mhz and a simulation of 2237 cycles which resulted in a throughput of $512 * 130/2237 = 29.75Mbps$. This implementation also used less area. This is shows that the synthesis from verilog to FPGA also impacts performance and area. It demonstrates that one can dramatically improve an algorithm by improving the underlying hardware.

E. Legup discussion

The Legup implementations were each incrementally better than other but not better than the pure software implementation from section III. To investigate why, we used the RTL viewer tool from quartus to see inside the model generated by Legup. We can see in figure 7 that it implements each function as a module. Looking further into a module was not fruitful since the RTL view gets messy as seen in figure 8. Only when looking into the generated verilog can we see that legup uses predefined functional units which bloat the system. For example, the verilog design has a dual port RAM for every array in the C code. This adds to the area since the C code does not need to access all the arrays in parallel. Moreover, it requires many cycles due to the numerous memory accesses. In the next section, we take advantage of this knowledge to create an improved hardware design.

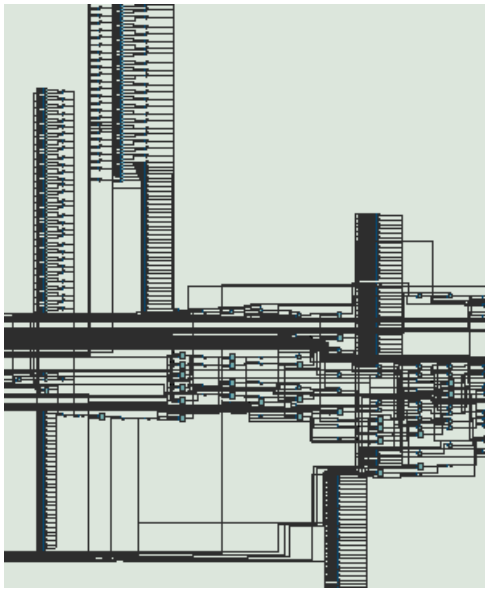

Fig. 8. The RTL view inside the SHA-256 module

TABLE II
TEST CASES TO VERIFY CORRECTNESS OF SHA-256 ALGORITHM

Flow	Throughput	Area
Software	6.32 Mbps	-
Hardware	25.21 Mbps	6133
Hardware Parallel Cyclone IV	19.42 Mbps	5959
Hardware Parallel Cyclone V	29.75 Mbps	2294

V. HARDWARE

The hardware design of the algorithm was implemented on a Cyclone V board using verilog. This design implemented the algorithm directly from specification [3] to verilog. A high level design diagram is seen in figure 9. The design shows a control block controlling two functional unit that haves access to a RAM. This ram is where the inputs are stored and is also where the outputs will be stored. The control logic controls the sequence of computation such that the data is first padded and then processed by the round computation unit. The RAM block holds space for the initial data block (512 bits) and the final output digest (256 bits). The RAM is a synchronized single port RAM which reads or writes 1 byte at a time. This makes it easy to read and write ascii strings.

The padder unit takes in as input the length of the input from the control block. Using that information, it writes the pad to RAM assuming that data in the RAM starts at address 0. It then signals to the control block that it is done.

When the padder has completed its task, the control block signals the round computation unit to begin. The round computation computes some of the data initialization during when the reset signal is high. It then initializes the scheduling vector (W) by reading elements from the memory. Unlike the Legup implementation, the scheduling vector (W) is stored locally within registers. Next it runs through the inner loop

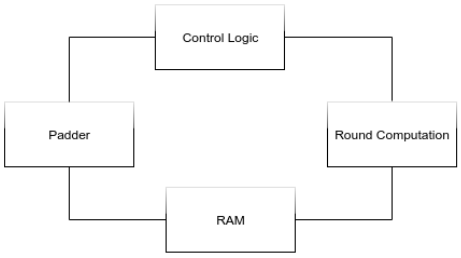

Fig. 9. Parallelizing the inner loop

computations and lastly it writes the computed hash values to RAM. The round computation unit also has a ROM which stores the constants of the SHA-256 algorithm. This idea was adopted from [5] and reduces the need to access RAM memory thus speeding up the computation.

This design went through an iterative design process where some techniques from the literature [2] [7] were attempted in order to increase the throughput. The rest of this section will go through some of the design choices in each iterative step.

A. Loop unrolling

The first design choice was to unroll the inner loop. The design from [2] unrolled the innerloop twice. Unrolling a loop lengthens the critical path, however, if the critical path is not located in the loop then we can unroll the loop and see an increase in throughput because we are doing more work in one clock cycle. For our first design we fully unrolled the inner loop (64 times unrolled). A fully unrolled loop finished in very few clock cycles but also had a very slow clock. The resulting throughput was a meagre $11.63Mbps$ which can be seen in table III. The second design had the loop unrolled twice and saw a jump in throughput to $215Mbps$. The third design had no loop unrolling and had a throughput of $248Mbps$. From these results it is clear that the critical path is shorter than two unrolled loops and therefore no unrolling is best.

B. Pipelining

Pipelining can also reduce the critical path. The design in [7] uses pipeining to speed up the compuation of the scheduling vector. In our design we recreate this block as seen in figure 10. We create a Fifo which stores values of the scheduling vector. We also create three extra registers which contain partial computations of the values of the scheduling vector. Figure 11 shows how the scheduling vector computation can be broken down into three additions that occur at three different time steps. The result from this section can also be seen in table III. From this implementation we achieved a throughput of 298 Mbps. That result is more than twice as fast as the software only implementation, ten times faster than the best legup implementation and faster than the implementation in [5]. From the results we can see that there is no improvements in the number of cycles required for the computation but there is a major gain in the clock frequency. This indicates that the scheduling vector calculation was part of the critical path and pipelining reduced it.

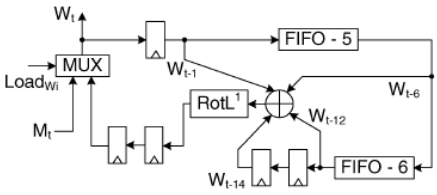

Fig. 10. Fifo based data initialization block from [7]

```
WtPre[0] = SIG1(fifo[1])+fifo[8];
WtPre[1] = SIG0(fifo[14])+WtPre[0];
WtPre[2] = fifo[15]+WtPre[1];

W[t] = WtPre[2];
```

Fig. 11. Breakdown of the scheduling vector additions

C. discussion

In this section we implemented the SHA-256 algorithm directly from specification to verilog. Then applied various techniques from the literature to speed up our results.

One attribute all these designs have is that they consistently use less area than the Legup synthesis. This can be attributed to the lack of bloat-ware. For example, this design uses exactly enough RAM for one block of input and one hash value. Reducing area footprint can be a potential for future research for the Legup HLS.

The lessons learned are that loop unrolling is inefficient if it increases critical path. Also pipelining will reduce critical path but not the number of cycles required to compute an answer. To further improve this design we can modify the RAM. A RAM that reads and writes 8 bits at a time also places a bottleneck since the rest of the algorithm uses 32 bit chunks of data. We can surely improve the number of clock cycles required to process the data if we were able to load in 32 bits at a time. Moreover, we can try to compute several SHA hashes at once as in [2], since our padder unit is left idle when the round computation unit is active.

VI. Conclusion

This project explored three different implementations of the SHA-256 algorithm in order to learn about HLS tools and application specific hardware design. First, a software only design was implemented and profiled as a benchmark to compare against. Second, using Legup, a HLS tool, a second design was synthesized. This design didn't fair very well against the software only design. It demonstrates that simply putting an algorithm onto hardware does not automatically speed it up. Lastly, a fully hardware design was implemented. This design had a higher throughput than the software design, all of the Legup designs and the design in [5]. Since legup is a HLS tool which can synthesize any application code, the hardware design demonstrates that an application specific design tends to have better performance and area metrics than a HLS tool.

TABLE III
Test cases to verify correctness of SHA-256 algorithm

Technique	Cycles	Fmax	Throughput	Area
Fully unrolled loop	158	3.59Mhz	11.63 Mbps	2054
Twice Unroolled loop	191	80.55 Mhz	215.92 Mbps	944
No Unrolling	223	108.15	248.3 Mbps	912
Pipelining	223	130.19 Mhz	298.91 Mbps	631

References

[1] B. Schneier, "Applied cryptography–protocols, algorithms, and..." 1994.
[2] H. E. Michail, G. S. Athanasiou, V. Kelefouras, G. Theodoridis, and C. E. Goutis, "On the exploitation of a high-throughput sha-256 fpga design for hmac," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 1, p. 2, 2012.
[3] P. FIPS, "180-4. secure hash standard," *National Institute of Standards and Technology*, vol. 17, 2015.
[4] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.
[5] N. Sklavos and O. Koufopavlou, "Implementation of the sha-2 hash family standard using fpgas," *The Journal of Supercomputing*, vol. 31, no. 3, pp. 227–248, 2005.
[6] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind," in *Tools for High Performance Computing*. Springer, 2008, pp. 93–113.
[7] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Cost-efficient sha hardware accelerators," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 8, pp. 999–1008, 2008.