SHA-3: FPGA Implementation of ESSENCE and ECHO Hash Algorithm Candidates Using Bluespec

Michel Kinsy, Richard Uhler
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA

Abstract

NIST has opened a public competition to develop a new cryptographic hash algorithm, and after the first round various algorithms have emerged as possible candidates. In this work, we have implemented on FPGA two of the candidates from the first round that are not known to be broken at this time, namely, ESSENCE by Jason Worth Martin and ECHO by the Cryptographic Research Group at Orange Labs, France Telecom. These algorithms convert a variable length message into a shorter message or hash that can be used for digital signatures, message authentication, and other applications. This report presents our design of these algorithms, coded in Bluespec and implemented on Altera's Cyclone II FPGA.

1 Introduction

ESSENCE is a Merkle-Damgård based hash algorithm, where an arbitrary-length message is hashed into a fixed-length output, specified by the user. To perform its hashing function ESSENCE breaks the message to be hashed into a series of equal-sized blocks, called Merkle-Damgård Block (MD Block), of one megabyte each.

Each block is run through an MD block computation unit, and hashed independent of other MD blocks, using a Merkle-Damgård based iterative construction. The MD block computation unit can have one or more compression function logics. The output data from one MD block is compressed with the outputs of previous MD blocks in the message. An initialization vector is used in the case of the first MD block of the message. The final hash is the compressed output of the padded last MD block, which contains the length of the data being hashed, the hash parameters, and a constant value final block . Figure 1 illustrates the overview of ESSENCE per author's description.

The other hashing algorithm implemented is ECHO. It is an AES (Advanced Encryption Standard) based algorithm that takes a message and a salt as inputs and produces a hash of any length from 128 to 512 bits. ECHO also uses the Merkle-Damgård construction in its compression approach. ECHO is designed with simplicity and security in mind. The compression unit is the only major building block in ECHO, which makes the system analysis very simple. ECHO also uses the same block cipher based approach seen in AES, so the design can profit from the AES security soundness. Figure 2 illustrates the hashing methodology outlined by its authors.

2 ESSENCE Design

ESSENCE consists of taking the message to be hashed and breaking it into MD blocks, where each MD block is hashed using the compression function and joined with other blocks to form a running hash.

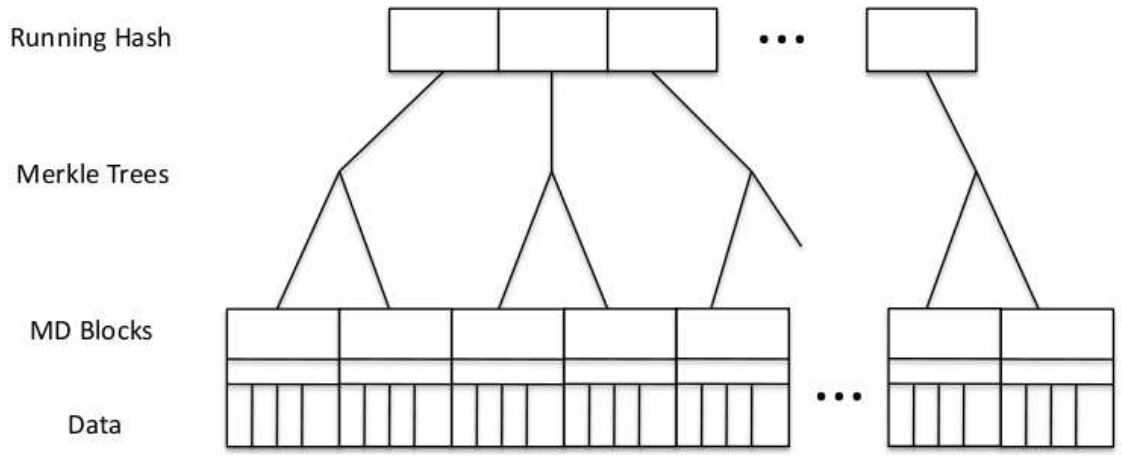

Figure 1: Overview of ESSENCE Hierarchy.

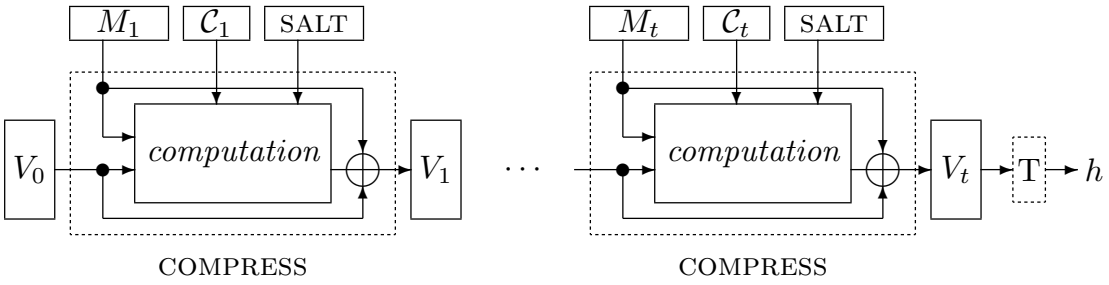

Figure 2: Overview of ECHO Composition.

2.1 ESSENCE Top View

Figure 3 shows our top level micro-architectural decomposition of the ESSENCE hash algorithm. Although this illustrating figure shows four MD blocks, our Bluespec implementation parameterizes the number of MD blocks actually instantiated.

Below is the Bluespec description of the ESSENCE module interface. The CONTROLLER_CMD type is a structure which contains information about the message starting address, length of the message to hash and the hash length.

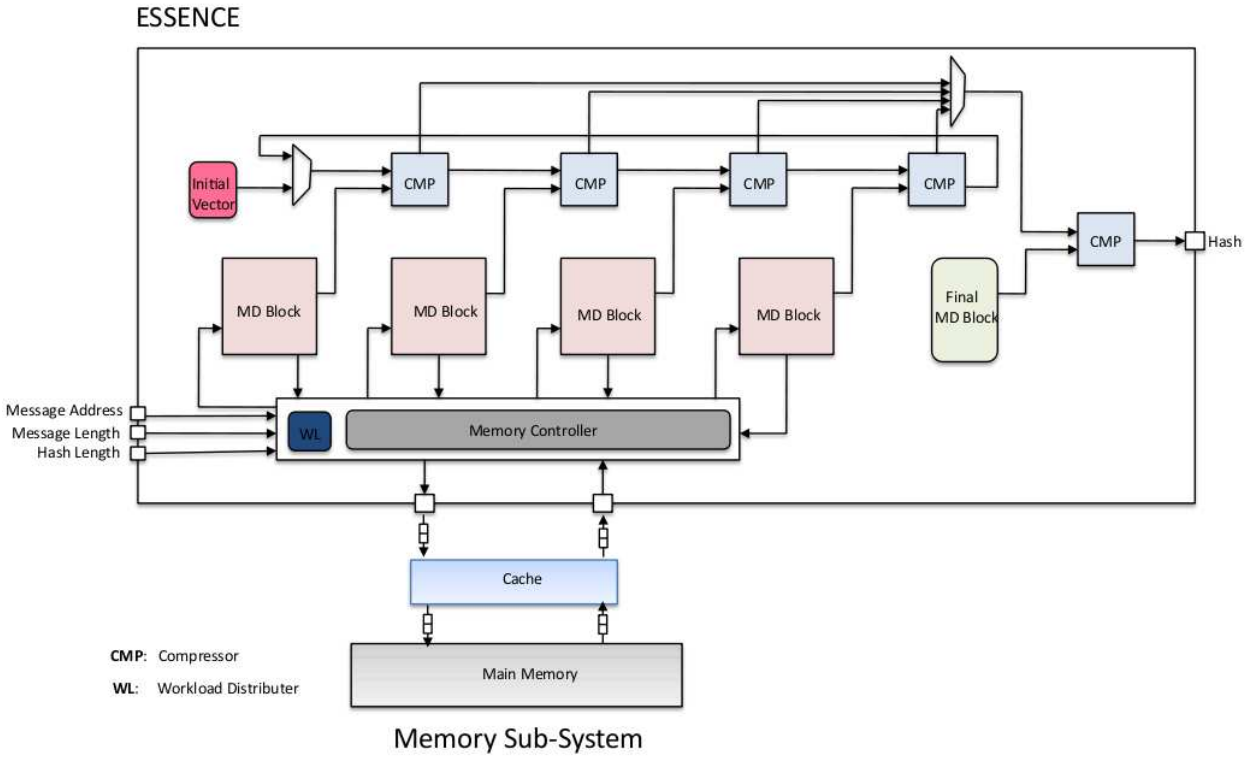

Figure 3: ESSENCE Top Level Micro-architecture.

```
interface Essence;
    method Action hash(CONTROLLER_CMD data);
    method ActionValue#(Hash) hash_output();
    interface Client#(DataReq,DataResp) mem_client;
endinterface
```

2.2 MD Block Design

Figure 4 shows the micro-architecture of the MD block, and below is its corresponding Bluespec description interface. The MDBLOCK_CMD type is similar to the CONTROLLER_CMD type, but in addition has information about the block number being compressed.

```
interface MDBlock;
    method Action hash(MDBLOCK_CMD data);
```

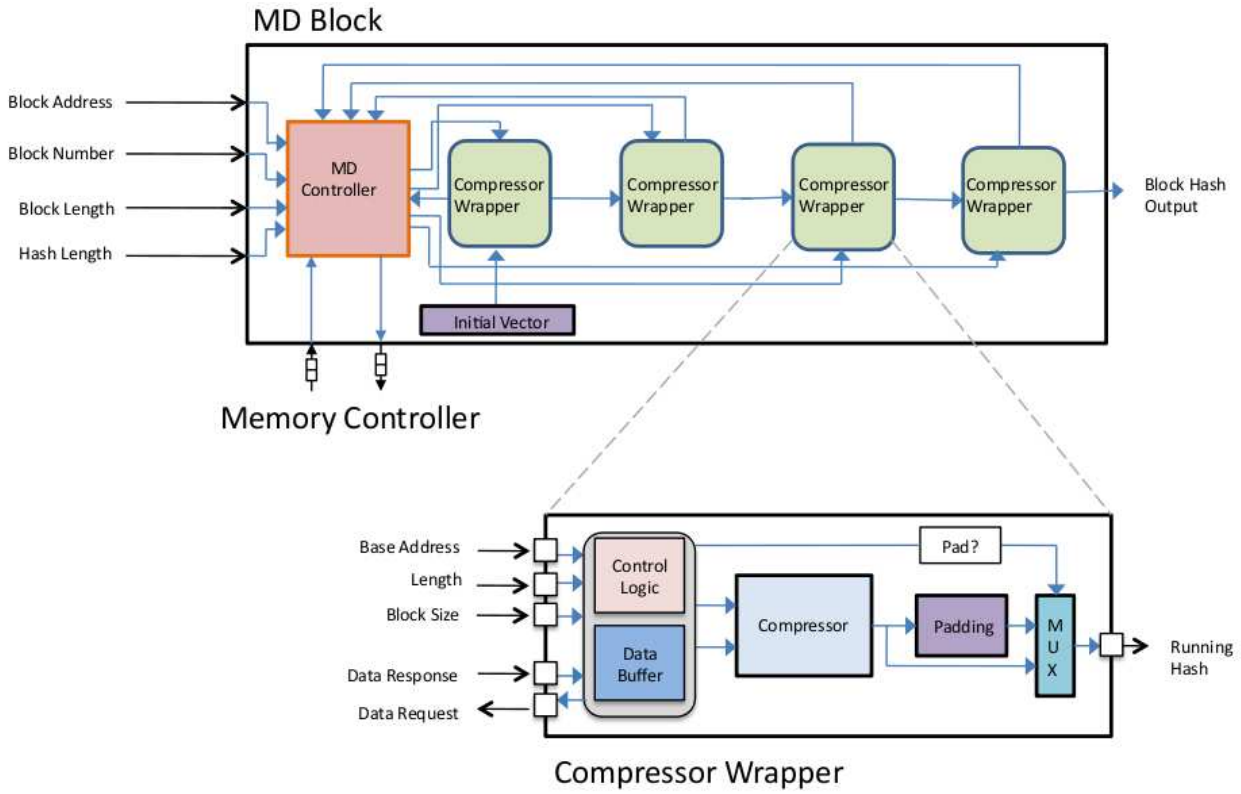

Figure 4: MD Block Micro-architecture.

```
    method ActionValue#(DATA) result();
    interface Client#(DataReq,DataResp) mem_client;
endinterface
```

Part of Figure 4 shows the compressor wrapper, which encapsulates the compression function found inside of the MD block. An MD block can have one or more compressor wrapper(s). Therefore, allowing the user to instantiate a variable number of compressor wrappers per block and per design.

```
interface EssCompressWrapper ;
    interface Client#(DataReq,DataResp) mem_client;
    interface Put#(EssCompressCmd)      work_request;
    interface Put#(DATA)                chain_input;
    interface Get#(DATA)                chain_output;
    interface Get#(DATA)                final_output;
endinterface
```

In our design, the MD block workload is divided among the different compressor wrappers, and each compressor wrapper makes its own memory requests via the MD controller independently from other wrappers.

```
interface MDController;
    interface Client#(DataReq,DataResp) mem_client;
    method Action init_hash (CONTROLLER_CMD hashData);
```

```
    method ActionValue#(EssCompressCmd)  getWork (CompresserID cid);
    method Action mem_req (DataReq ir, CompresserID cid);
    method ActionValue#(DataResp) mem_resp (CompresserID cid);
endinterface
```

2.3 Compressor Module

The core of the ESSENCE Hash function centers around its compression function. The message hash ultimately is constructed by compressing message data blocks together with constants and intermediate compression results.

The compress function takes an input block of 16 words and compresses that down to a block of 8 words. The compression is achieved through a fixed number of successive permutations with linear and feedback functions as illustrated in figure 5.

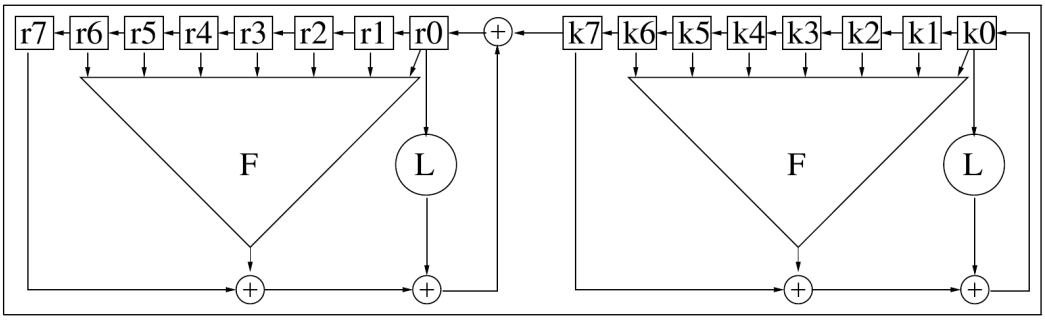

Figure 5: ESSENCE Compression Logic (From ESSENCE Specification)

F is a boolean formula which takes 7 booleans as input. In the compression logic F is applied to the nonboolean inputs in a bitwise fashion. If the seven inputs to F are labeled a through g, then F = $abcdefg + abcdef + abcefg + ...$, where multiplication is performed with the AND operation, addition is performed with the XOR operation, and ... includes a large number of additional terms very similar to the first.

L is a linear function which operates on a single word by representing the word as a row vector of bits and multiplying that vector by a known boolean matrix. The choice of matrix is such that the matrix multiply can be achieved with a feedback shift register in Galois configuration.

To support message hash lengths between 128 and 512 bits, there are two versions of the compress function. For hash lengths between 128 and 256 bits the compression word size is 32 bits, and for hash lengths between 256 bits and 512 bits the compression word size is 64 bits. The only other difference between the two compression functions is the specific matrix used in the linear function L.

2.3.1 Compression Design

For the initial design of the ESSENCE Compression function the focus is on correct operation rather than high performance operation. To allow for flexibility later on when we are more concerned with performance, all module interfaces are latency insensitive. This is done simply by using Bluespec's Server interface for each module. For the initial implementation we assume requests for all modules are served in order.

The compression function is broken down into two instances of permutation logic, each instance of which instantiates a single L function and F function, resulting in the four Bluespec modules as illustrated in figure 6.

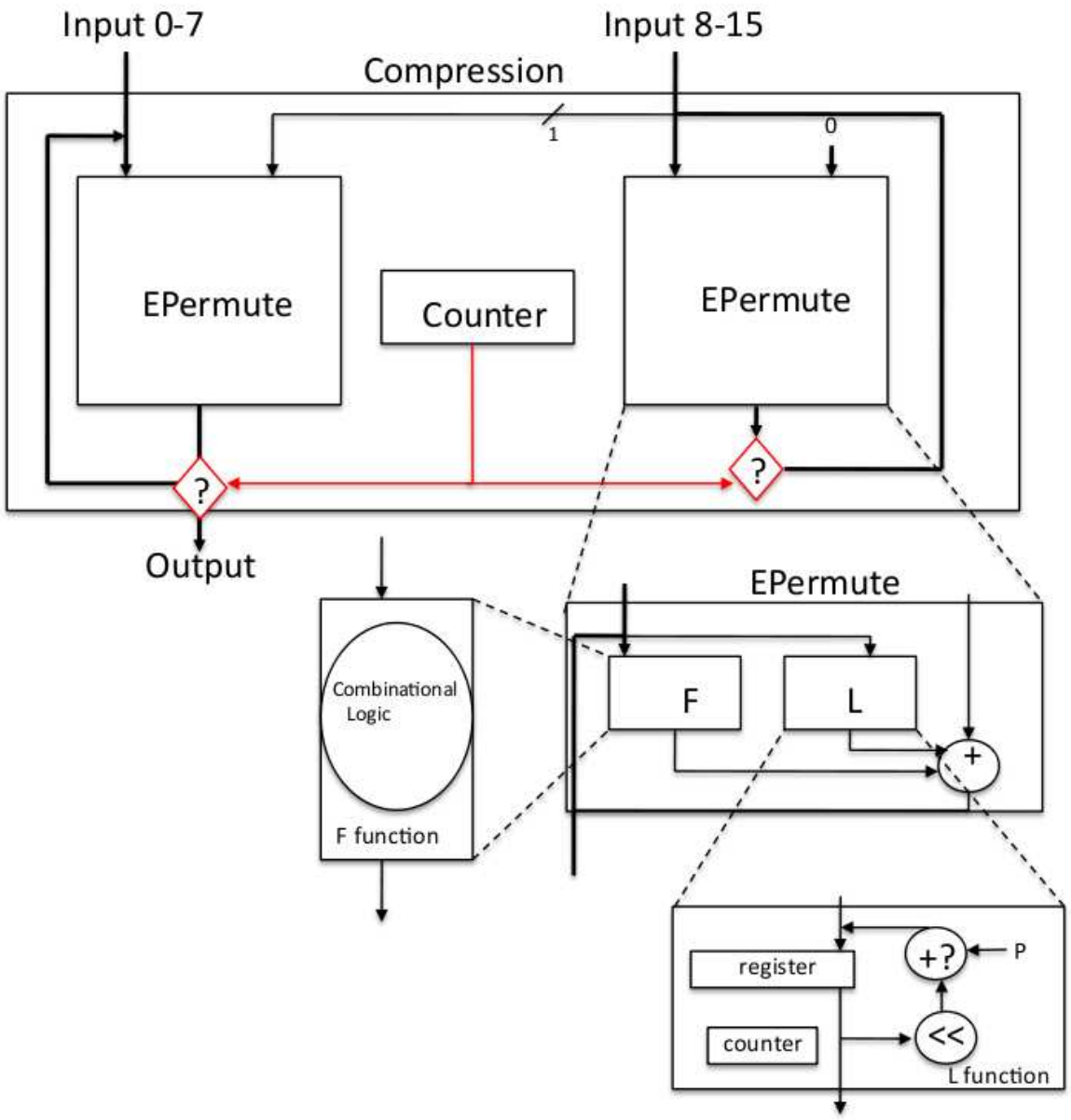

Figure 6: Compression Architecture.

The top level module performs the permutation a fixed number of times, using a counter to keep track of the iterations. The permutation passes its input to the feedback function and linear function then sums the result for one word of the output and permutes the input to obtain the rest of the output. The linear function uses the Galois shift register configuration to implement the matrix multiply, using a counter to keep track of how many shifts have occurred. Finally, the feedback function is implemented as pure combinational logic.

2.3.2 Sharing 32 Bit and 64 Bit Compression

Except for the word sizes and number of rounds and XOR condition in the L function, the 32 bit and 64 bit compressions are exactly the same. In anticipation of changes we may desire to make to improve performance and save area, rather than instantiating two different compression functions we have a single compression function which can do either the 32 bit compression or 64 bit compression. This works by using 64 bit words for all the interfaces and an additional flag which says whether to do the 32 bit compression or the 64 bit compression. When doing the 32 bit compression, half of each word will be ignored.

The logic for the top level compression, permutation, and feedback function modules can be shared as is across the different sized compressions. For the linear function we instantiate two different modules and choose the appropriate one to use depending on the compression size.

3 ECHO Design

The ECHO hash function takes a message and a salt as inputs and produces hash output of length between 128 and 512. The message is broken into small data blocks of 1536 or 1024 bits, and each block is hashed using either compress512 or compress1024 compression functions depending on the specified hash output length. The resulting data from one block is joined with other blocks by using it as the chaining variable for the next message block.

3.1 ECHO Top View

Figure 7 shows our initial top level micro-architecture decomposition of the ECHO hash algorithm in which we were trying to parameterize the number of compression blocks that can be instantiated. But in general, there seems to be no advantage in having more than one compression function block. The algorithm itself is very sequential, requiring the result from a previous message block compression before the next one can begin. Therefore our final hardware implementation uses only one compression as shown on Figure 8.

Below is the Bluespec description of the ECHO module interface. The `EchoCompressCmd` type is a structure which contains information about the message starting address, length of the message to hash, the hash size, and the salt.

```
    interface Echo ;
        method Action hash(EchoCompressCmd data);
        method ActionValue#(Hash) hash_output();
        interface Client#(DataReq,DataResp) mem_client;
    endinterface
```

One key component of the ECHO module is the data fetcher unit. This unit plays two critical roles, which are memory data fetching and manipulation(to present the compression function the data in the appropriate byte-structure format), and padding of the tail of the message.

```
    interface EchoDataFetcher ;
        interface Client#(InstReq,InstResp)  mem_client;
        interface Put#(EchoCompressCmd)      compress_request;
        interface Get#(MessageBlock)         message_block_data;
    endinterface

typedef struct {
    Bit#(64) counter;
```

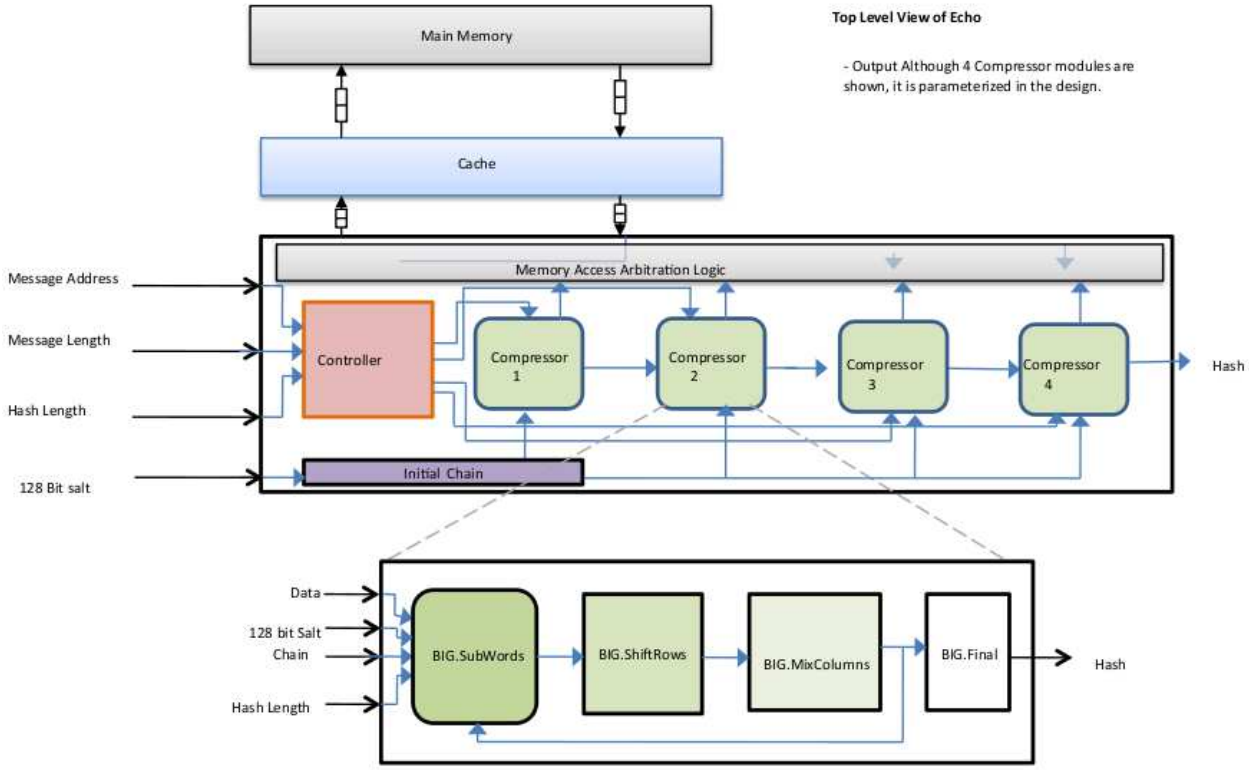

Figure 7: Initial Top Level View of ECHO.

```
    Salt salt;
    CompressData data;
} EchoDataCompressRequest deriving(Bits, Eq);

typedef Server#(EchoDataCompressRequest, Chain) EchoDataCompress;
```

3.2 ECHO AES Design

One of the prime design goals for ECHO was to reuse a lot of existing work, including AES. ECHO uses whole blocks from the AES algorithm as is or only slightly changed. In this document whenever we refer to AES, we mean ECHO's specific AES block. References to the real AES algorithm as described in the fips-197 spec and not associated with the ECHO hash algorithm will be made explicit.

AES operates on a 128 bit word and 128 bit key to produce a new 128 bit word. It is convenient to view the 128 bit word as a 4x4 array of 8 bit bytes called the AES State. The AES operation, then, consists of a sequence of transformations on the state: sub bytes, shift rows, mix cols, and add round key.

The sub bytes transformation does a byte by byte substitution exactly as described in the AES fips-197 spec.

The shift rows transformation simply shifts each of the 4 rows in the state by a different amount, also as described in fips-197.

The mix cols operation does a column by column matrix multiplication using a simple matrix where a special multiplication is performed as described in fips-197.

Add Round Key does a byte by byte XOR of the state with the 128 bit key input to AES. Add Round Key differs from the fips-197 specification in that the keys are inputs to the AES algorithm instead of generated

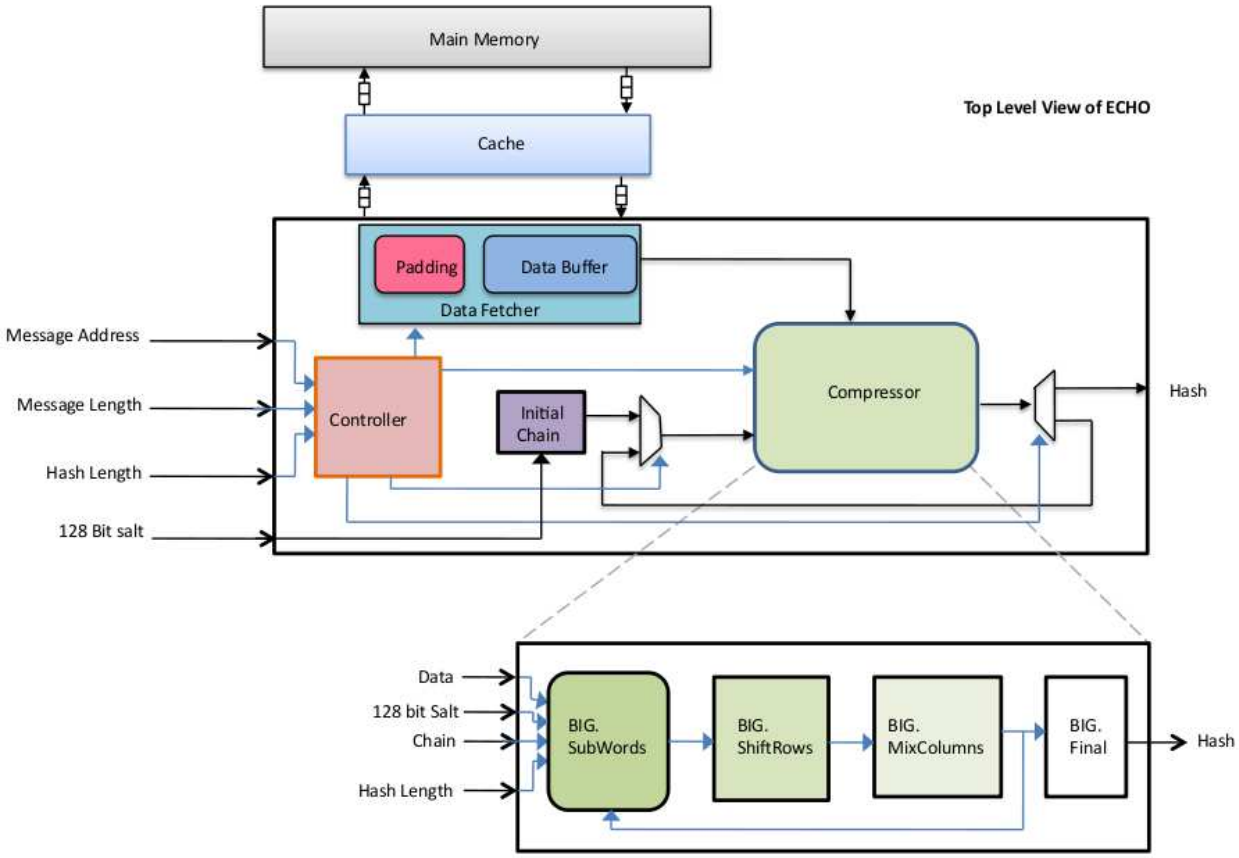

Figure 8: Implemented Top Level View of ECHO.

through a key expansion, and there is a separate 1 byte key for each byte of the AES state rather than a 1 byte key for every column in the AES state. This difference is not made clear in the ECHO spec, but is apparent from the reference implementation.

3.3 Decision to Implement AES

There are already verilog implementations of AES available which we could have used in our project by wrapping them in Bluespec. Instead we chose to implement the portion of AES that ECHO requires from scratch in Bluespec.

It's been suggested having an implementation fully in Bluespec will be much more convenient than a Bluespec wrapped verilog implementation. The ability to easily use Bluesim through the entire design is one example.

While ECHO claims it uses AES as is, in reality there are slight variations. These include a modified AddRoundKey transformation and no Key Expansion phase. To use an existing AES implementation we would have to extract those specific pieces we need, modify how they are used slightly, and integrated it with a custom AddRoundKey transformation, which seems excessive given the simplicity of the AES ECHO needs.

Implementing AES in Bluespec, with full control over the interfaces and design, gives more opportunity to practice design and understand how different design decisions affect performance and area of the implementation. That there already exists AES implementations in verilog gives the added benefit of a baseline to compare against to know if our implementation is excessively limited in any ways.

3.3.1 Design

Like other parts of our design, the initial design targets correctness, not performance. All interfaces are of Bluespec's Server type so they are latency insensitive. Initially we assume all requests are served in order.

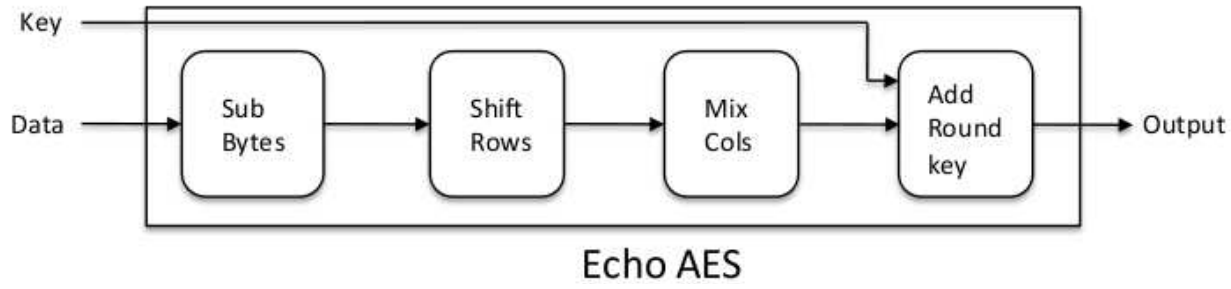

Figure 9: ECHO AES Architecture.

The implementation of the top level AES module trivially passes the AES State through the sequence of transformations as shown in figure 9. Because all the modules use a Server interface we can use mkConnection for plugging most of the modules together.

The sub bytes transformation was initially implemented by sequentially applying the substitution on each byte of the AES State. The byte substitute is implemented with a big Bluespec case statement. At this point we had little idea of how much area or delay is incurred with a big switch statement for the substitution. It turned out the case statement was converted into a RAM lookup, which fit fine on the fpga, so we eventually changed to performing the substitution for each of the 16 bytes in parallel as described in more detail in the performance analysis of AES.

The shift rows transformation is simply implemented using a bunch of Bluespec assignments.

The mix columns transformation mixes each of the four columns in parallel, taking a stage to perform all the needed multiplication and an additional stage for the sums.

The AddRoundKey is implemented as 16 parallel XORs.

4 Testing

At some point in the project it became clear that making progress in our implementation of the hash algorithms ECHO and ESSENCE depended significantly on having an easy way to verify our implementations were outputting correct results. We used a couple different strategies for testing, one for the compression modules which didn't access memory, and one for the top level hash modules, which accessed memory.

4.1 Testing Compression Modules

The compression modules for ECHO and ESSENCE were both broken down into a number of smaller modules which all made use of Bluespec's Server interface. Each module took some small bits of data as a request and after some number of cycles output some different small bits of data as a response.

To get started as simply as possible we used Bluespec's StmtFSMs to drive inputs into the modules and verify the outputs of them. Each module had its own StmtFSM test driver. Inputs were all hard coded and often picked at random. Expected outputs were also hard coded and based on either our understanding of what the output should be from the specs, or generated by running pieces of the c reference code on the inputs and copying over the outputs. If the outputs from our Bluespec modules matched the hardcoded expected outputs, we printed out using display statements that the test passed, otherwise we printed that the test failed.

When tests failed, the usual strategy for debugging was to apply display statements in key parts of the code to figure out what was going on. It was usually clear then what the problem was.

When the compression modules passed all tests in simulation we modified the test driver to output the test results over the CBus so we could run the tests on the FPGA. This worked fine, except it took a surprising amount of time (around 2 hours) to synthesize all the tests. It was suggested using StmtFSMs were likely contributing to that. Because we primarily ran tests in simulation, we didn't worry about StmtFSMs taking a long time to synthesize.

4.2 Testing with Memory

The modules the next level up in the design hierarchy from the Compress modules added a new challenge to testing, which was they had access to memory. It was no longer sufficient to just hard code inputs in StmtFSMs and check outputs, because we needed to provide data through memory.

We had access to Bluespec source code for running a simple MIPS processor on our FPGAs. The infrastructure for that processor included support for specifying memory via a mem.hex file, specifically for use in providing the instructions of the program the processor should run. We reused that same infrastructure as is for this project, using the mem.hex file to supply data instead of instructions.

Initially we started out with a rather ad hoc testing strategy. We would manually load the mem.hex file with some data and hard code in Bluespec the hash length, message length, and message address for a hash request to be performed. The expected output was again generated using the c reference code, which was setup to take a hash length and file as its data to hash. This strategy was easy to use and matched well our desire to get something up and running as fast as possible, but it also had some problems. If we wanted to try a different hash length or message length, we had to recompile the Bluespec code. Matching up the message length with the actual length of the file in the right units was error prone, leading to more silly recompilations.

Things worked really well when we managed to correctly give both the c code and the Bluespec code the same inputs, but the times when we made little errors in the test parameters were very frustrating because we would spend all this time trying to understand why our implementation wasn't working when really it was working fine and it was our tests which were broken.

We made a couple changes to our ad hoc strategy which were extremely helpful in debugging more productively. First, we supplied all of the hash request parameters through the mem.hex file instead of just the message data. This way we wouldn't have to recompile the Bluespec code to change the message length or hash length. Second, we wrote a script which takes the same command line arguments as the c reference code and automatically generates the correct mem.hex file from those arguments.

With our changes in place it was simple to compile the Bluespec once, then provide different hash request parameters in the same format to both the script we wrote and the c reference code and verify using diff whether the outputs matched. Once that was up and running the bulk of our debugging time was focused where it should be, on the details of our implementation of the hash algorithms, such as padding.

As we encountered files, message lengths, or hash lengths that didn't hash properly we added them to a suite of tests which could all easily be run with a single command.

4.3 A Data Generator

Eventually we got around to testing our hash implementations on larger files and quickly ran into trouble. The infrastructure for the simple processor which we were using for memory access didn't support very much data. Rather then put a lot of effort into researching and setting up a new infrastructure to support multi-megabyte files, we wrote a simple data generator.

The data generator acts like memory from the perspective of the hash implementations. Given a memory address it returns a word of data some number of cycles later. The data returned is generated deterministically from the memory address through an arbitrary function.

We also implemented the data generator function in c and plugged it into the c reference code so we could verify our implementation was consistent with the reference.

5 FPGA Implementation and Performance Analysis

In this section, we present the resource utilization of the two hashing algorithms as well as some of the different implementation refinements done to improve their performance.

5.1 ESSENCE FPGA Implementation Details

From the timing report, the requested frequency of the final ESSENCE implementation synthesized is 80.3 MHz and the estimated frequency is 68.2 MHz. The design's critical path is from the MD controller unit [Figure 4] to the compressor wrapper units back to the controller. The requested period is 12.46 nanoseconds and estimated period is 14.66 nanoseconds due to the worst slack of 2.2 nanoseconds.

Figure 10 shows the final FPGA floorplan of our ESSENCE design. Table 1 presents the summary of the data collected during analysis and synthesis of the design.

Table 2 shows the resource utilization per module for the design. Resource usage is shown in an inclusive matter, for example, the ESSENCE hardware unit uses 39377 sequential elements from which 20367 of these elements are allocated to the MD Block unit. (See Figures 3, 4 and 6 for the modules composition).

FIFOs in the design also consume a fair amount of the resources allocated to the design. And because of time constraints, no significant hardware design refinement is made to reduce these FIFOs or to improve the cycle time. The linear function, in the EPermute module, uses a shift register but an alternative is to use a single matrix multiplication which takes less cycles.

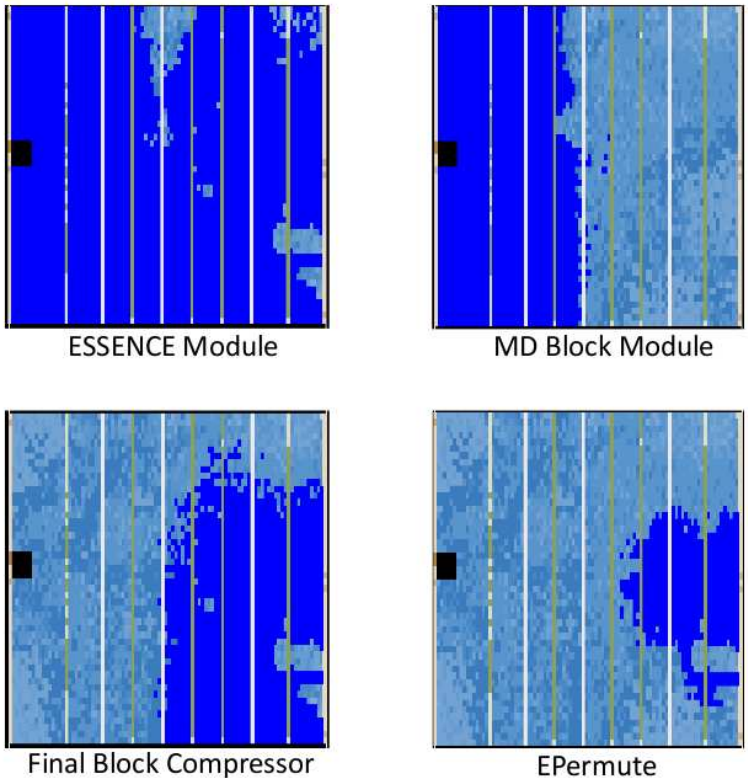

Figure 10: Cyclone II FPGA Floorplan for ESSENCE

Table 1: Synthesis Environment and Summary for ESSENCE

Analysis and Synthesis Summary	
Quartus II Version	8.1 Build 163 10/28/2008 SJ Full Version
Revision Name	smipsV2System
Top-level Entity Name	smipsV2System
Family	Cyclone II
Total memory bits	849,920
Embedded Multiplier 9-bit elements	0
Total PLLs	0
Resource	**Usage**
Total combinational functions	61476
Logic element usage by number of LUT inputs	
4 input functions	33671
3 input functions	24354
$\leq$ 2 input functions	3451
Logic elements by mode	
normal mode	60916
arithmetic mode	560
Total registers	
Dedicated logic registers	42401
I/O registers	0
I/O pins	8
Total memory bits	849920
Maximum fan-out node	clk_0
Maximum fan-out	42604
Total fan-out	339429
Average fan-out	3.26

Table 2: Hardware Elements Usage by the Major Components in ESSENCE

Module	**Sequential Elements (Registers)**	**Combinational Elements**
smipsV2System	42401	61476
mkCoreFPGA	41701	59962
ESSENCE	39377	56875
MD Block	20367	28320
Compressor Wrapper	18713	26372
Compressor	13711	20982
EPermute	5172	8168
Linear Function	775	926
Feedback Function	1032	2957

Table 3 shows number of cycles it takes the compression unit in ESSENCE and its submodules to compress a message block to 512 bits or 256 bits hash.

Table 3: Number of Cycles per Module per Hash Length

Module	Cyles per 512 Request	Cycles per 256 Request
Compressor (EssenceCompress)	2640	1216
EPermute	66	36
Feedback Function	1	1
Linear Function	64	34

Figure 11 shows that the number of cycles taken to hash a message is proportional to the message length. Shorter hash lengths also take more cycles to compute.

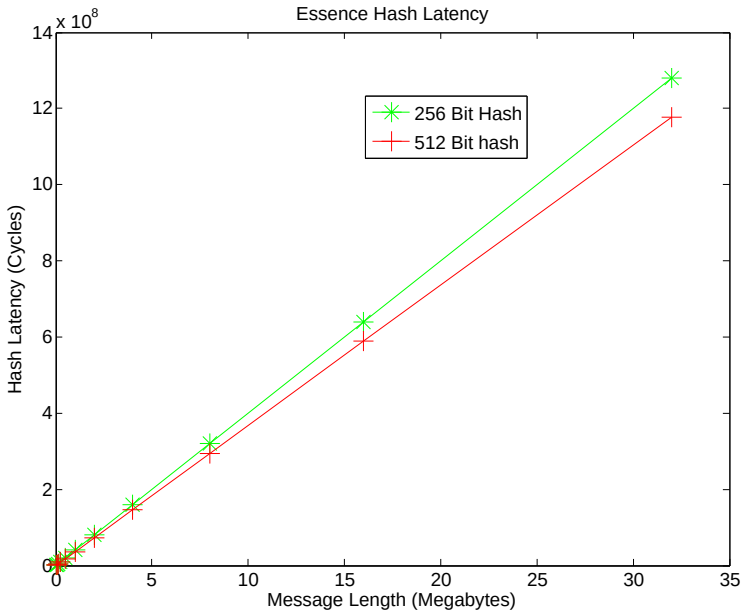

Figure 11: Number of Cycles per Message Length and hash Length.

5.2 ECHO FPGA Implementation Details

From the timing report, the estimated frequency of the final ECHO implementation synthesized is 70.6 MHz with estimated cycle time or period of 14.16 nanoseconds.

Figure 12 shows the final FPGA floorplan of our ECHO design. Table 1 presents the summary of the data collected during analysis and synthesis of the design.

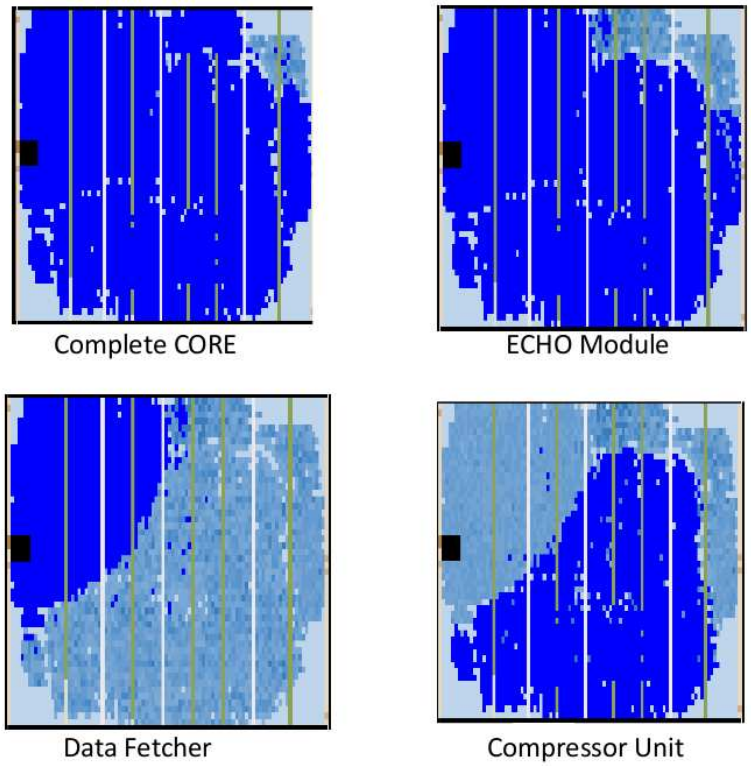

Figure 12: Cyclone II FPGA Floorplan for ECHO

Table 4: Hardware Elements Usage by the Major Components in ECHO

Module	LC Combinationals	LC Registers	Memory Bits
smipsV2System	40601	20802	915456
CoreFPGA	39091	20102	119808
ECHO	36055	17726	65536
EchoDataCompress	17699	12370	65536
BigFinal	1860	1576	0
BigMixCols	3778	2055	0
BigShiftRows	1604	2163	0
BigSubWords	6105	4452	65536
EchoDataFetcher	16450	3804	0

Table 3 shows number of cycles it takes the compression unit in ECHO and its submodules to compress

a message block to 512 bits or 256 bits hash.

Table 5: Cylce Counts for Major ECHO modules.

Module	Cyles per 512 Request	Cycles per 256 Request
EchoDataCompress	341	273
BigMixCols	1	1
BigShiftRows	1	1
BigSubWords	32	32
BigFinal	1	1
BlockFetch	1280	1920

The critical path goes through the padding operation at the end of the hash. This involves figuring out how many bits are in the last block and choosing the appropriate padding scheme to use for the block.

Figure 13 shows that the number of cycles taken to hash a message is proportional to the message length. Unlike the ESSENCE case, longer hash lengths do lead to more computation cycles.

5.3 On Area and Performance

When we initially synthesized ECHO for the fpga the compression alone was on the order of 70,000 LC combinatorials, which is more than our fpga held. Our implementation strategy was to start as simple as possible, making all modules latency insensitive. This led us to using FIFOs all over the place, most of which were unnecessary or larger than necessary. By removing unneeded FIFOs and making other FIFOs sized with 1 element instead of the default 2 the area of the echo compression reduced drastically.

Even when ECHO was small enough to fit on the fpga, however, we ran into problems with local routing congestion. We traced this down to the BigSubWords module, which was sequentially passing each of 16 128 bit words in the compress state through a single AES module. To reduce the congestion we introduced an additional AES module so that each AES module was shared by only 8 words instead of 16. This brought the local routing congestion down just enough so that with fitting optimizations turned on the design fit on the fpga.

The performance of our ECHO implementation initially was lousy. The AES module, for example, took 60 cycles to complete a single request. We didn't have much time to play around with the design to improve performance, but with what little time we did have, we increased the performance drastically.

The problem in AES was that we had made everything latency insensitive, assuming it took any number of cycles more than 1. This was good for getting a working implementation up and running, but hurt when it came to putting together a bunch of small operations each taking more than 1 cycle. By changing that assumption and replacing our Server modules with Bluespec functions, we were able to do all of AES in a single cycle down from 60 cycles.

One interesting phenomenon we observed in a few places was that sometimes by replicating logic the overall area goes down. For example, instead of using a single MixCols module time multiplexed to mix 4 different columns we used 4 MixCols modules, one for each column. This reduced the number of cycles for that operation by a factor of 4, and it didn't increase the area at all. It may be by specializing each column to its own MixCols module we could do away with complex arbitration logic. Perhaps also by turning the MixCols module into a function and removing a synthesis boundaries the tools were able to do a better job optimizing the logic.

The performance of the best performing implementation we managed before running out of time to do more optimization is listed in tables 4, 5. Most of the time for compression is spent in the BigSubWords

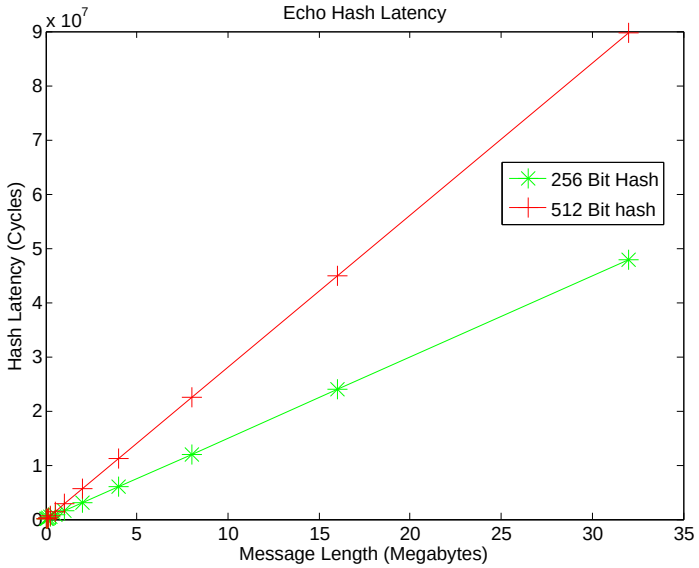

Figure 13: Number of Cycles per Message Length and hash Length.

module. This module passes each of 16 words through two full rounds of AES. We tried replicating the AES logic 16 times so the BigSubWords operation would only take a few cycles, but it didn't fit on the board. The problem, perhaps surprisingly, wasn't the combinational logic or registers from AES, but rather the memory usage. AES performs sixteen 256 element table lookups, which the synthesis tools put into RAMs on the fpga. The fpga only has 250 of these RAMs, and 16 AES instantiations would require 16*16=256 RAMs, so it didn't fit.

Another observation is even if we could fit 16 AES instantiations on the board it wouldn't improve the time it takes to hash messages from file data. Our memory setup is pretty appalling, and wasn't really the primary focus of our project. As a result we can compress data at a faster rate than we can fetch it. Table 5 shows it takes us 341 cycles to compress 1024 bits of data for 512 bit hash lengths, but 1280 cycles to fetch that 1024 bits. It's even worse for 256 bit hash, which takes less time to compress but actually compresses more data each time.

6 Conclusion

In summary , both ECHO and ESSENCE are implemented in hardware directly from the algorithm description and reference documentation. In our FPGA implementation, we use Altera Cyclone II board as the targeted FPGA environment, and Bluespec as the hardware design language. The final ECHO design has an estimated frequency of 70.6 MHz and consumes 39091 combinational elements. ESSENCE uses 61476 total combinational function elements and run at 68.2 MHz.

Due to time constraints, we performed very little performance improvement on ESSENCE. For example, a single matrix multiplication, instead of shift register, could be used for the linear function to significantly reduce the number cycles.

Memory data retrieval also was not fully investigated. We did not for example examine the data caching behavior in the case of multiple MD blocks. Also the usage of on chip data generator doesn't allow for complete exploration of the data fetching overhead for each hashing algorithm .

The reference to ESSENCE is [3] and the reference to ECHO is [1]. Other relevant documents are [2, 4].

References

[1] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. *SHA-3 Proposal: ECHO*. Technical Report http://crypto.rd.francetelecom.com/echo/, Orange Labs, France Telecom, 2009.

[2] Ivan Damgård. A design principle for hash functions. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.

[3] Jason Worth Martin. *ESSENCE: A Candidate Hashing Algorithm for the NIST Competition*. Technical Report http://www.math.jmu.edu/ martin/essence/Supporting_Documentation/essence_NIST.pdf, James Madison University, 2009.

[4] Ralph C. Merkle. One way hash functions and des. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 428–446, London, UK, 1990. Springer-Verlag.