EVALUATION OF THE HARDWARE PERFORMANCE SPACE OF SHA-3 CANDIDATES BLUE MIDNIGHT WISH AND CUBEHASH USING FPGAS

by

Robert Lorentz

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

KGaj
Dr. Kris Gaj, Dissertation Director

Dr. Jens-Peter Kaps, Committee Member

Dr. Qiliang Li, Committee Member

Dr. Andre Manitius, Department Chair

Dr. Lloyd J. Griffiths, Dean, Volgenau School of Engineering

Date: 12/09/2011

Fall Semester 2011
George Mason University
Fairfax, VA

Evaluation of the Hardware Performance Space of SHA-3 Candidates Blue Midnight Wish and CubeHash Using FPGAs

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Robert Lorentz
Bachelor of Science
George Mason University, 2006

Director: Kris Gaj, Associate Professor
Department of Electrical and Computer Engineering

Fall Semester 2011
George Mason University
Fairfax, VA

DEDICATION

I dedicate this Thesis to my entire family for the support, patience, understanding, and advice that has allowed this research to be performed.

ACKNOWLEDGEMENTS

I would like to thank all of the people who made this Thesis possible.

I especially thank Dr. Kris Gaj for providing excellent guidance and inspiration and for sharing his endless enthusiasm for cryptography and for FPGA design, without which I would have never undertaken this effort or developed such an interest in these subjects. I also would like to thank Ekawat "Ice" Homsirikamol for his endless patience and support as I adapted his earlier and ongoing research for use in my experiments, without which my research could not have been completed in such a thorough and successful manner. Also, I would like to thank Marcin Rogawski for his advice and assistance regarding my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

EVALUATION OF THE HARDWARE PERFORMANCE SPACE OF SHA-3 CANDIDATES BLUE MIDNIGHT WISH AND CUBEHASH USING FPGAS

Robert Lorentz, MS

George Mason University, 2011

Thesis Director: Dr. Kris Gaj

In 2007, the National Institute of Standards and Technology (NIST) announced a public competition to develop a new cryptographic hash algorithm to become the SHA-3 standard. This algorithm should allow flexibility in the design tradeoff decisions between performance and circuit area. This study evaluated two SHA-3 Round 2 Candidate Algorithms, Blue Midnight Wish and CubeHash, to define their performance space in FPGA hardware. High throughput designs were created using multi-message techniques, and single-message Basic Iterative and Folded techniques were applied to find designs of relatively low area. The results show a large performance range for both algorithms, but the fine granularity achieved with parallel cores of CubeHash is superior to the inflexible pipelined architecture of Blue Midnight Wish.

CHAPTER 1: Introduction

1.1. Cryptographic Hash Function

In today's world of digital commerce, telecommuting, and international business there is an incredible and constant amount of reliance on cryptography to facilitate the lifestyle that has become the standard. Not only are people living these high-tech lifestyles, but also they are doing so with the ubiquitous computing model - a wide variety of new types of mobile devices have become pervasive in our environment and the expectation is that almost all everyday tasks can be performed with these devices.

One specific type of cryptography relied upon for security within these tasks is the cryptographic hash function. Conceptually, a hash function can take any piece of message or data and give a short piece of data that identifies this input message. Ideally, it should be infeasible to find another input message that will produce the same hash function output. If someone receives a piece of data that they have a known good hash output value for, they can verify that the received data has integrity by running the received data through the hash algorithm and comparing it to the known good value. An application for this would be to ensure that a file transfer has completed successfully.

Most commonly, a hash function is used along with as a digital fingerprint to provide cryptographic authentication services. An example of this would be if Alice writes a letter to Bob, creates a hash value *Hash(M)* of the message, and then encrypts

this using a public key encryption system using her private key corresponding to her public key. This example can be expressed as in equation (1.1).

$$A, M \,||\, E(Apr, Hash(M)), B \qquad (1.1)$$

The term used for this technique is a digital signature. This scheme implements authentication because Bob receives and reads the message *M*, decrypts *E(Apr, Hash(M))* using Alice's public key, and then uses the same hash algorithm locally on *M*. It is infeasible to find another message *M'* that results in the same hash value *Hash(M)*, so if the hash value of *M* matches the received *Hash(M)* then it can be said with certainty that Alice did in fact send this message. The reason for the public key encryption algorithm is to prove that *only* Alice could have sent the signature due to the properties of a public key encryption system, which are beyond the scope of this paper. The reason for sending a signature using *Hash(M)* for authentication instead of the entire message *M* is that it is extremely inefficient to use the entire encrypted *M* since it will double the storage space and data transfer bandwidth utilization. In contrast, the value of *Hash(M)* is a very small constant size regardless of how large *M* becomes. Due to the properties of hash functions, this scheme is theoretically very close to the security of using the entire encrypted *M*, and in practical terms it is identical.

Speaking formally, a cryptographic hash function takes an input message *M* of arbitrary length, performs a series of cryptographic transformations on this data, and

results in a hash message digest *H* of a length defined by the algorithm. Unlike other cryptographic functions such as symmetric key cryptography, hash functions do not utilize a change-able key. Since no secret key is needed for the algorithm, the resulting message digest *H* for a given *M* can be computed with no knowledge except the public specification of the algorithm.

The length of *M* can be arbitrarily long, and the size of *H* is of a fixed small size, so a hash function is a many-to-one function that maps the domain *M* to the range *H*. Being a many-to-one function, there are three cases possible for each given value *h* of *H*. First, the function may be unable to associate any values with a given *h* value of *H*. Secondly, the function may associate one value with a given *h*. Finally, the function may associate up to *Length M* values with a given *h*. The ideal situation is to have an even mapping of values such that every given *h* in the Range of the function has an equal number of possible messages that can map to it. If too few values map to a given *h*, then it effectively reduces the keyspace. If too many values map to a given *h*, it becomes easier to find a collision. These situations reduce the security properties of a cryptographic hash function and are undesirable.

Formally speaking, there are two security requirements of a cryptographic hash function. First, given *Hash(M)* it must be computationally infeasible to find *M*. If it were possible to do this, then a signature could be used to compromise an encrypted message. Secondly, it must be computationally infeasible to find a message *M'*, different than *M*, such that *Hash(M') = Hash(M)*. If this was violated, then a collision would be said to occur, and a malicious message *M'* could be used. The impact of this could be

making it appear as if another party signed something that they did not agree to, or reducing the confidence of a true signature by pointing out alternate values.

1.2. Motivation

The extreme reliance on cryptographic services makes it crucial that they are secure. If the security of all commonly used algorithms were compromised, there would be a catastrophic impact to the global economy, military and government security, and to many people's lives.

Therefore, the National Institute of Standards and Technology (NIST) standardizes cryptographic algorithms for use. These standards are often what are adopted by commercial, government, and for general use. The previous algorithms standardized by NIST, using the prefix Secure Hash Algorithm (SHA), have been SHA-0, SHA-1 and SHA-2. The United States National Security Agency (NSA) developed all three of these algorithms internally, with the most recent SHA-2 standard being published in 2001.

Attacks have been found against the strength of these hash algorithms. Significantly, in 2005 researchers found [1] an attack against SHA-0 that can find collisions in an unacceptably short amount of time. Similar attacks have been found [2] against SHA-1 to the extent that NIST has recommended that as of 2010 SHA-1 use be discontinued in favor of SHA-2. With SHA-2 being the only standardized algorithm not yet broken, NIST has called an open competition for a completely new SHA-3 algorithm [3]. This open competition for SHA-3 mimics the earlier move by NIST to call for open

competition for the new Advanced Encryption Standard (AES) symmetric key cryptography algorithm to replace the Data Encryption Standard (DES) algorithm that had significant weaknesses exposed. When we talk about an algorithm being broken, it means that it has become feasible for a well-funded organization such as a national security agency to be able to find a collision within a reasonable amount of time such as weeks, instead of hundreds of years or longer as should be the case.

For these SHA-3 candidate algorithms, the hardware performance and flexibility is of major interest to NIST [4], with NIST stating that several algorithms were deemed unacceptable due to area requirements in hardware, and that other algorithms were seen as better due to providing fine-grained control of the amount of parallelization that they could be designed with. Therefore, research that further explores the performance space of hash algorithms is clearly valuable to those evaluating or designing algorithms.

Blue Midnight Wish and CubeHash were two of the fourteen algorithms to advance to the second round of the NIST SHA-3 competition. Neither of these algorithms made it to the final third round of the competition, but they are well-documented and interesting algorithms to research, hence they have been chosen for this work. These algorithms also have fundamentally opposite properties in terms of hardware performance. Blue Midnight Wish is very large but can be pipelined many times to gain performance. In contrast, CubeHash cannot be pipelined, but is so small in terms of area that it could be replicated many times on the same FPGA to form parallel processing units.

1.3. Blue Midnight Wish

The SHA-3 competition round 2 version of the Blue Midnight Wish Algorithm specification [5] has been used for this work. This hash algorithm, proposed by researchers from the Norwegian Institute of Science and Technology, has been designed to be much more efficient than the current SHA-2 standard while also offering the same or improved security.

There are variants of Blue Midnight Wish for message digest sizes 224, 256, 384, and 512. For this research, the 256 and 512 bit variants were examined. The significant difference between these four variants is the size of the data buses and operations, there is no significant different to the flow of data or cryptographic operation.

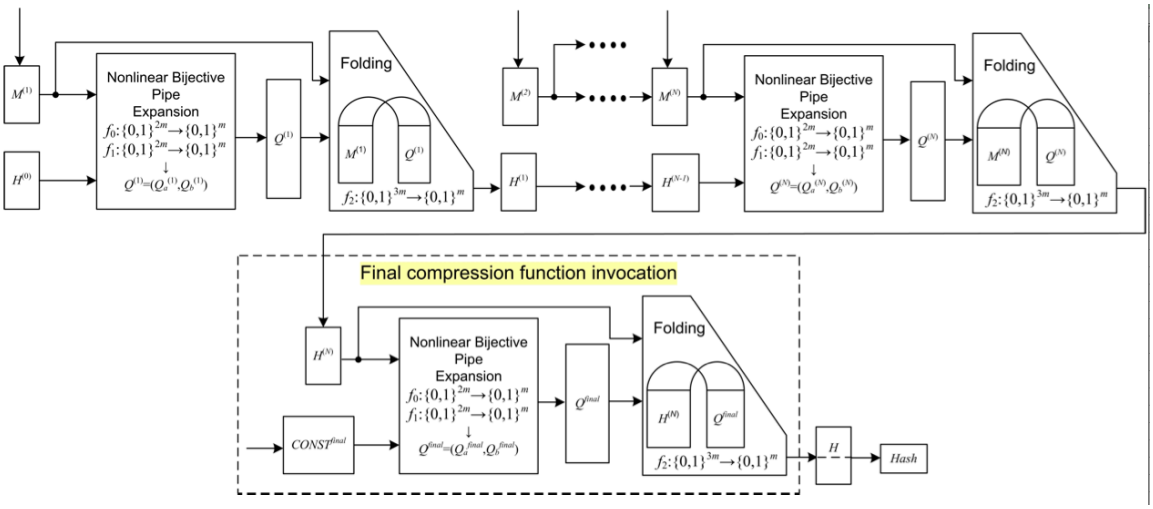


Figure 1.1: Graphical Representation of the Blue Midnight Wish Hash Algorithm [5]

As shown in Figure 1.1, each block of the message *M* and previous hash value *H* is processed through two non-linear bijective pipe transform function *F0* and double pipe expansion function *F1*, which creates a quadruple pipe *Q*. Subsequently, a folding operation *F2* is performed using input of *M* and *Q*. The result is the next *H,* which is then either processed in the same manner along with the next block of the message *M*. If there is no more *M*, it is run through a finalization compression function invocation of the datapath using a constant value in place of *H* and *H* in place of *M*.

By using the quadruple pipe *Q* as suggested by Joux [6], Blue Midnight Wish is protected from generic multicollision attacks because it's chaining pipe is at least twice the size of its message digest size. Multicollision attacks are of significant concern in cryptanalysis since they can lead to collisions occurring in practical situations with iterative hash functions. Unfortunately for the hardware implementation of Blue Midnight Wish, dealing with a large chaining pipe comes at the price of significantly increased area.

1.4. CubeHash

The SHA-3 round 2 version of the CubeHash specification [7] has been used for this work. This hash algorithm, proposed by Daniel J. Bernstein of the University of Illinois at Chicago, is very different from the current SHA-2 standard as well as from most other cryptographic hash functions.

The main design goal of CubeHash is simplicity. Instead of going with more traditional modes of hash algorithm modes of operation, CubeHash foregoes this in favor

of a very simple series of add, rotate, swap, and xor operations done iteratively in identical rounds. There are tunable options *r* and *b,* where *r* sets the number of rounds (multiplied by 10) and *b* sets the size of the blocks. Due to the tunable variables, there are several variants of CubeHash. These variants are notated in the form "CubeHash *r*/*b*-*MDSize*." The variants used in this study were CubeHash16/32-256 and CubeHash16/32-512.

Due to the usage of only very few operations, which are not at all large in terms of area required on the physical device, CubeHash implements to a very small area on hardware. The critical path is also very short due to the tight round loop comprised of fast operations, so a physical implementation of the algorithm achieves high clock speeds. Another interesting property of CubeHash is that its internal operation is unchanged based on whether it is the 256 or 512 variant; this parameter simply indicates how much of the final output is discarded. This final output is computed by outputting the first *h*/8 bytes of the internal state as the message digest.

This study compares the 256 and 512 variants of the CubeHash algorithm. A better examination of the performance space of CubeHash would be to examine what happens when *r* and *b* are set to a range of values. However, the author indicates in the specification that once the algorithm is standardized (presumably with parameters *r* and *b* similar to those we've chosen), these parameters will be firmly set and the only variation in the practical application of the algorithm will be to change the message digest size. Therefore, it makes sense to examine the algorithm in this sense even though it is fundamentally disadvantaged in many of our calculations for 256 bit message digests.

One interesting attribute of CubeHash, due to the way it handles 256-bit and 512-bit message digest versions, is that one dedicated circuit can be configured to operate in either 256-bit or 512-bit hash operation based on a single input pin that is configurable in real-time. Since there is no difference in the main hash core, the only change would be in how input and output circuitry is configured. This is in contrast to Blue Midnight Hash, which would require significant modification to the datapath for such functionality.

1.5. Previous Work

Due to the NIST SHA-3 competition, there have been many papers and results published regarding all of the candidate algorithms, including the ones we are interested in. Some of this previous work has been the foundation of this research, and some of it has been duplicated in an attempt to compare the results within the framework of this research and compare it to other techniques.

There has been an experiment [8] that created a Multi-message pipelined version of Blue Midnight Wish for evaluation against single-message a version. This pipelined version started from a baseline version similar to our Basic Iterative version of Blue Midnight Wish. The result, although not explicitly named this, was an x1-PPL18 version that was replicated in this work. The reason for replicating the earlier effort was to compare the architecture within the ground rules of this work, and also to experiment with further alternative architectures.

A very low area implementation of Blue Midnight Wish was created using the concept of a very basic set of primitives implemented along with a set of instructions

read from memory [9]. Using this type of method can obtain extremely low area utilization, but at the extreme cost of performance. This approach is so different than the others that it was not included in the evaluation of performance space being performed in this paper.

Although no papers describing a folded hardware design of Blue Midnight Wish are known to exist, there has been research done [10] towards decomposing the algorithm further to primitive pieces than is described in the algorithm's specification or obvious from simple analysis. The decomposition was done to aid in cryptanalysis, but this is also very useful knowledge for creating alternate designs. The decompositions described in the previous research correspond to the folded /32(h) design described in this thesis.

CHAPTER 2: Design Methodology

The goal of this thesis is to explore the overall performance space of Blue Midnight Wish and CubeHash, and also to compare them in a fair manner with each other. Certain ground rules must be adopted to ensure the fairest comparison is performed. The ground rules established in [11] were used as a guideline for this work. The rules are summarized as follows:

- FPGAs are used as the hardware implementation target.
- Uniform input/output interface are used for all designs.
- The same basic building blocks are used in the implementation of all designs. Using these same primitives provides maximum consistency between the designs and allows for more fair comparison of designs.
- The source code for all designs is written in VHDL.
- The same assumptions and simplifications have been made such that no design gains an unfair advantage in comparison to another. The assumptions are that external implementation of padding is performed, that extremely high-speed data interface buses up to and including FIFOs are present and external to the design, and that the designs are saturated by incoming messages to be hashed.

- The CAD tools used for each FPGA device family are the most recent versions of commercial tools that are supplied by that device's vendor. Using the vendor supplied tools and not using a competitor or third party's tool ensures that there is not a negative bias being applied.
    - *Xilinx:* Xilinx ISE Design Suite 13.2
    - *Altera:* Quartus II v. 9.1 Subscription Edition
- No special dedicated hardware resources of a chip are used, including Block RAM, Memory Bit, DSP unit, or Multiplier resources. Using these resources makes it extremely difficult to compare designs since these resources are not standardized between vendors or even device families.
- Identical and easy to repeat tool options are accomplished by using the ATHENa (Automated Tool for Hardware EvaluatioN) software package, developed at George Mason University [12].
- Results have been gathered for several FPGA devices, spanning the most popular two vendors Altera and Xilinx. For both Blue Midnight Wish and CubeHash, for a specific message digest size such as 256 bit, the same device was used. This allows fair comparison within the results obtained for the specific algorithm, so that a meaningful comparison can be made between designs of one algorithm as well as between the two algorithms. The *Stratix III* family was selected for Altera and the *Virtex 5* family was selected for Xilinx. These families are of comparable technology. The devices specifically chosen from these families were chosen due to their general-purpose nature, and the smallest device that could accommodate

the largest design was used. The fastest speed grade was selected because that option increases the headroom of the results.

  - *256 Altera:* ep3sl70f780c2
  - *256 Xilinx:* xc5vlx85ff1153-3
  - *512 Altera:* ep3sl110f1152c2
  - *512 Xilinx:* xc5vlx220tff1738-2

- Generalized design is used for all designs, as shown in section 3.1.
- Designs are verified using a testbench along with Known Answer Test (KAT) test vector files provided as part of the hash algorithm's SHA-3 competition submission package. The KATs that were used to verify the designs include various lengths of messages for short single block, multiple blocks, and multiple segments. This gives strong confidence that the designs are all correct.

The deviations that have been made from the guideline rules are summarized as follows, along with a rationale for each deviation. In general, the deviations have been made to allow an exploration of the entire performance space to be performed and to allow exploration of both single-message and multi-message techniques.

- The optimization target differs based on the primary application of interest for the technique. Folded designs optimize based on area, Basic Iterative designs optimize for a balance between area and speed, and Multi-Message designs optimize for throughput. This differs from the guidelines to optimize for throughput to area ratio, because doing so would not give a comprehensive assessment of the entire performance space.

- A uniform protocol is used for all Single-message designs, and a uniform protocol is used for all Multi-message designs. These protocols differ from each other because a message ID must be added for Multi-message designs to ensure proper correlation between input messages and output message digest hash values.
- A uniform testbench is used for all Single-message designs, and a uniform testbench is used for all Multi-message designs. These testbenches differ from each other because testing of the message ID is required in Multi-message designs and data input to the circuit is handled differently as well.

Benchmarking of the performance is done with respect to Area, Throughput, and Latency. Depending on the application of the designer, one of these could be the most important design constraint. Area can be crucial when designing an architecture that must be on the most economical devices, or must coexist on a device that has other major functions. Throughput can be most important if designing for very high performance situations such as a network interface hardware component, or if expecting heavy traffic of very long messages. Latency, meaning the time it takes for a hash value to complete, is also very important. In some applications, specifically those with many small incoming messages, latency could be the most important design decision.

The measurement of Area for each of the designs is given in terms of utilized Slices for Xilinx Virtex 5 family of devices, and ALUTs for Altera Stratix III family of devices. The reason for the difference is that these vendors do not use the same terminology or internal device architectures that would allow. Therefore, using these area measurements allow comparison of devices to each other, but comparisons of area

are not equal. In contrast, throughput can be directly compared between vendors because that output is in a common unit of Megabits per second.

For each design in this paper an equation is given for Latency which is measured as Hash Time in Cycles, and Throughput which is measured in Megabits per Second. The general equations used to derive these specific equations are given in Table 2.1. In this table, r is the number of rounds required to complete one hash in the round structure, T is the clock period in microseconds, k is the factor of folding, n is the number of messages able to concurrently be in a multi-message architecture, and b is the block size of the function. Note that the clock period in microseconds T is a value that differs between all designs and is not a constant.

The most basic derivation is the Basic Iterative x1, where throughput is defined as the block size divided by the number of rounds required for a single message block multiplied by the clock period in microseconds. In other words, it takes r clock cycles that last T microseconds to result in the hashing of one block of size b bits. This equation is extended to Folded architectures by multiplying the denominator by folding factor k. The reason is that for each round r, it will now take k clock cycles instead of 1 clock cycle to complete it. The parallel multi-message architectures are similar to the Basic Iterative equation except the throughput is multiplied by n, the number of parallel units that are on the device. This is intuitive because there will be n times the output since all of those parallel cores operate in parallel. Finally, the pipelined multi-message architecture is similar to the parallel equation except it requires an additional n clock

cycles to process a single message block. Note that the pipelined equation actually simplifies to be the same as Basic Iterative when computing throughput.

Table 2.1: Generic Equations of Latency and Throughput

Architecture	Time required to process a single message block	Throughput
Basic Iterative x1	*Tblock = r*T*	*Tp = b/Tblock*
Folded by *k, /k*	*Tblock = r*k*T*	*Tp = b/Tblock*
Multi-message x1-PPL*n*	*Tblock = n*r*T*	*Tp = n * b/Tblock*
Multi-message x1-PAR*n*	*Tblock = r*T*	*Tp = n * b/Tblock*

2.1. Technique for Basic Iterative Designs

The Basic Iterative designs used in this exploration were the ones established in a previous George Mason University paper, and their published source code was used to generate results and to act as a baseline for the modified designs explored in this paper. The technique for creating a basic iterative design is to create an architecture that follows the description of the hash algorithm that is provided or implied by the algorithm's specification. Following this technique gives a balanced result simply denoted in the results listings as BMW x1 as it is assumed to be the baseline.

The goal of a Basic Iterative design is to find a balanced implementation of the algorithm. These designs are not typically going to be the fastest, or the lowest area; but they also won't be the slowest, or the highest area. It is useful to study balanced designs because they are representative of the performance of a typical implementation of the algorithm. It is also an interesting data point to compare different designs of the same

algorithm. As an example, it's hard to know how much area a folded design has saved or what impact that savings has had on speed unless there is a basic iterative benchmark to compare against.

2.2. Technique for Folded Designs

When creating a folded design, there are two different approaches that can be taken. First, the architecture can be folded horizontally. Horizontal folding is done when the same round is used repeatedly and the datapath width stays the same size as the original Basic Iterative x1 architecture. The alternative is vertical folding, which is achieved by splitting the datapath width in half and concatenating it back at the end of the folded round. In some circumstances, this is the only feasible approach. In this study, the folded architectures created were horizontally folded.

The reason for folding is to conserve physical area on the implemented circuit. Folding typically shortens the critical path of the circuit, resulting in an increased clock frequency. Despite the increased clock frequency, more clock cycles must occur for a block of the message to be processed. Therefore, it is typical to expect a lower throughput in a folded design, despite the increased clock speed. Similarly, latency is expected to become worse as a result of folding.

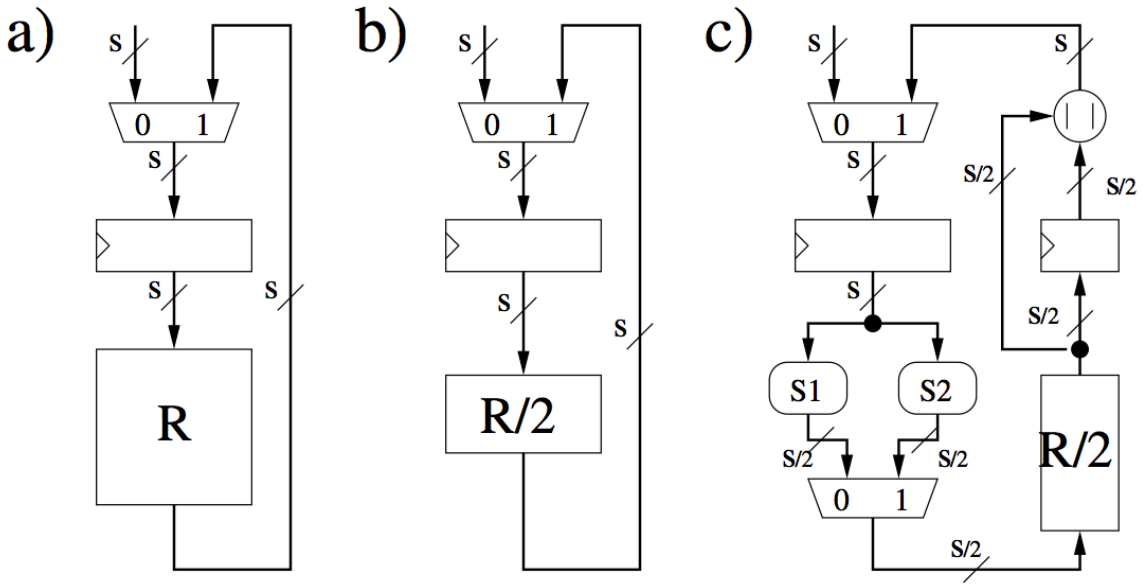


Figure 2.1: Three Hardware Architectures of a Hash Function. a) Basic Iterative: x1, b) Folded horizontally by a factor of 2: /2(h), c) Folded vertically by a factor of 2: /2(v). R – round, S1, S2 – selection functions [15].

2.3. Technique for Pipelined Multi-Message Designs

The technique to create a pipelined version of a basic iterative design is to examine the round and find segmentation between operations that occur in parallel. For example, if several additions occur in parallel and then their result moves on to be further processed, a natural point to pipeline the design will be the point between the completion of addition and the start of the next operations. This is physically achieved by registering all of the data. The associated control logic will also need to be modified. When segmenting the architecture in to pipeline stages, the overall approach should be to split it in to segments that have roughly the same critical path. If an architecture is converted in to an x1-PPL2 two-pipeline stage design, but one of the stages has a much longer critical path than the

other, the throughput of the design will suffer because the clock frequency will remain similar to the x1 architecture.

As noted in the discussion of the throughput equations, the pipelined equation simplifies to be the same as Basic Iterative when computing throughput. However, as the number of pipeline stages increases, the latency grows longer. When designing a pipelined architecture, this balance must be taken in to account. A designer needs to find the correct architecture that gets the best throughput while maintaining a latency that is as low as possible. In general, it may be tempting for a designer to greedily pipeline the design to a very high degree, and this method will usually produce shortest critical path and the maximum throughput. However, the side effect is that latency will be very long. An architecture is likely to exist that has a more modest number of pipeline stages that attains almost the same throughput while incurring much less latency penalty. In contrast, a design that has too few pipeline stages may be significantly slower than the maximum throughput possible to be obtained through the pipelining technique.

The overall operation of a Pipelined design is to accept data through FIFO units, have that data read in by FSM1 to a SIPO unit, then to have that SIPO drive one single datapath and FSM2. The FSM2 unit tracks state for all of the pipeline stages, and determines logic based on which timeslot of the pipeline is at which portion of the datapath. Communication between the single FSM2 and FSM3 is done to select when a PISO should capture the output from the datapath. The single FSM3 controls all PISO units and muxes them selectively through to a single output FIFO. By having a single FSM3, the results are able to output through one unified data interface. Using the

Message ID within the data protocol correlates the incoming FIFO data and the outgoing message digests.

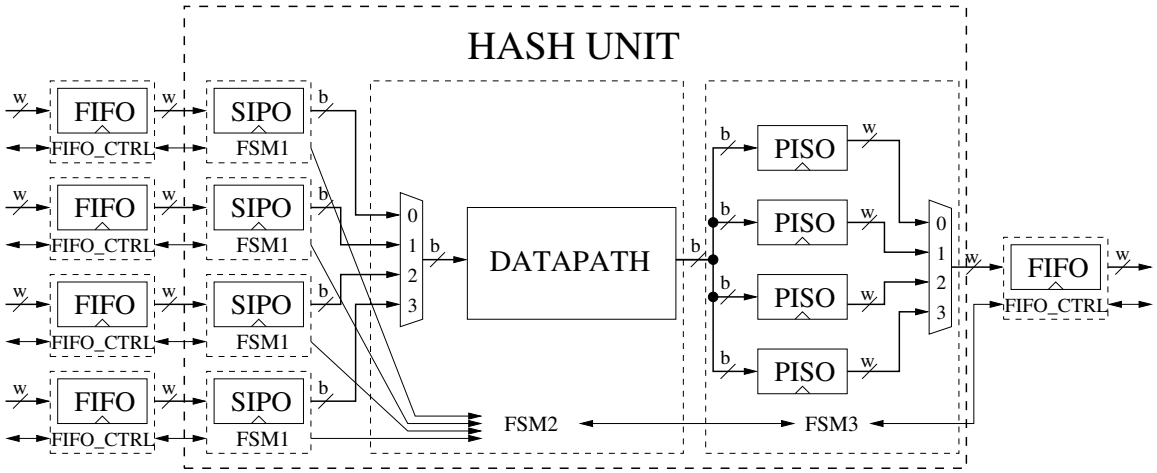


Figure 2.2: Detailed Overview of Pipelined Architecture [15]

2.4. Technique for Multiunit Multi-Message Designs

A Multiunit multi-message design results in several identical Basic Iterative hash cores being simultaneously placed on the same device. In general, the most suitable type of algorithm for a Multiunit design would be a lower area design. The reason is that it can be parallelized with finer granularity than a larger area design, and can therefore maximize the number of cores held on an FPGA device of a given area. In contrast, if each core was quite large, then there could be a significant amount of unusable area on a given FPGA device and that would not be ideal. For this practical purpose, the design goal should be to fill as much area as possible of a device with operational cores.

The overall operation of a Multiunit design is to accept data through FIFO units, have that data read in by FSM1 to a SIPO unit, then to have that SIPO drive one specific datapath and FSM2, and then driving the data in to a PISO. Up until this point, operation is exactly like a Basic Iterative design. All FSM2 units in the design communicate with a single FSM3 that controls all PISO units and muxes them selectively through to a single output FIFO. By having a single FSM3, the results are able to output through one unified data interface. Using the Message ID within the data protocol correlates the incoming FIFO data and the outgoing message digests.

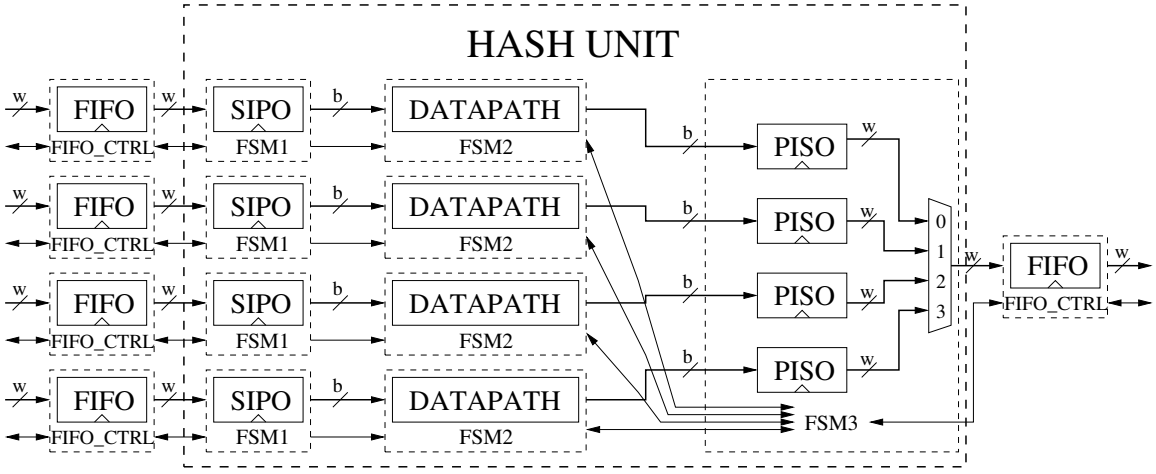


Figure 2.3: Detailed Overview of Multiunit Architecture

CHAPTER 3: Interface and Protocols

The generalized circuit design and uniform interface used in this study were adopted from earlier work done by Ekawat Homsirikamol as described in his thesis work [13], which describes the foundation for the George Mason University SHA-3 candidate algorithm FPGA implementation designs. This framework is general enough that it was found acceptable for all 14 SHA-3 Round 2 candidates.

3.1. Generalized Circuit Design

The basic concept of the generalized circuit design is that data passes in and out from the Datapath one word at a time from external FIFO units. The Controller also accesses these FIFOs to inform them when to read and write, and to analyze the FIFO status to know when data is ready to read or ready to be written. The controller and datapath then communicate with each other to accomplish the hash operation.

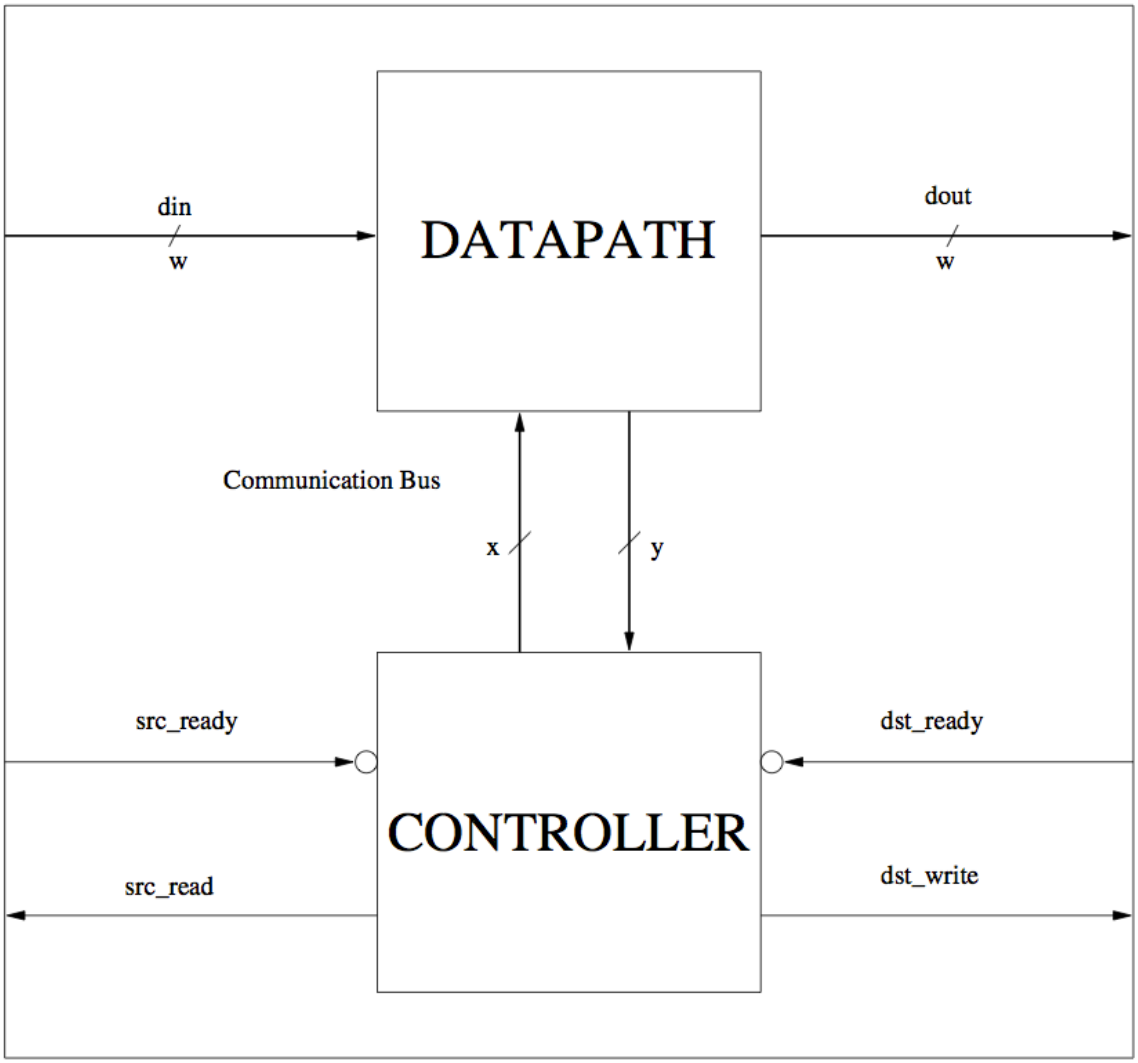


Figure 3.1: Generalized Top Level [13]

As each word of data comes in from the FIFO, the Datapath accumulates it in to a Serial In Parallel Out (SIPO) unit. Following hash operation, a Parallel In Serial Out (PISO) unit sends resulting data from the datapath to the output FIFO. Without these SIPO and PISO units, an entire block of a hash message would need to come in and out of the circuit at once. This would require *mw* number of input pins and *b* number of output

pins. Typically this is 8 times the quantity of pins, and creates an unrealistic physical requirement on the FPGA device used. Also, the assumption has been made that the data interface is extremely fast, so running a faster *io_clk* to clock the FIFO, SIPO, and PISO units is acceptable. Assuming a fast data interface matches closer to the reality of the physical FPGA devices, such as in the Xilinx Virtex 7 Family which includes a 12.5Gb/second serial interface standard [14] yet does not feature any significant increase in the number of exposed general purpose I/O pins.

The Generalized Controller consists of three Finite State Machines (FSMs) that generate control logic for the design. FSM1 is responsible for interfacing to the input FIFOs and loading that data in to the datapath's SIPO unit to prepare the incoming message to be input to the hash function. When a piece of the message is ready for the hash core, FSM1 sends control signals to FSM2 that work can begin. FSM2 is responsible for all control signals required for the specific hash algorithm in the design. FSM3 is responsible for taking a completed message digest from the datapath, loading it in to the PISO unit, and controlling that PISO to load its data in to the external output FIFO.

FSM1 and FSM3 are reusable parts that would only need to be modified in rare circumstances. FSM2 will always require extensive modification because it drives all control for a unique datapath, although the interfaces FSM2 shares with neighboring FSM1 and FSM3 will in general stay the same.

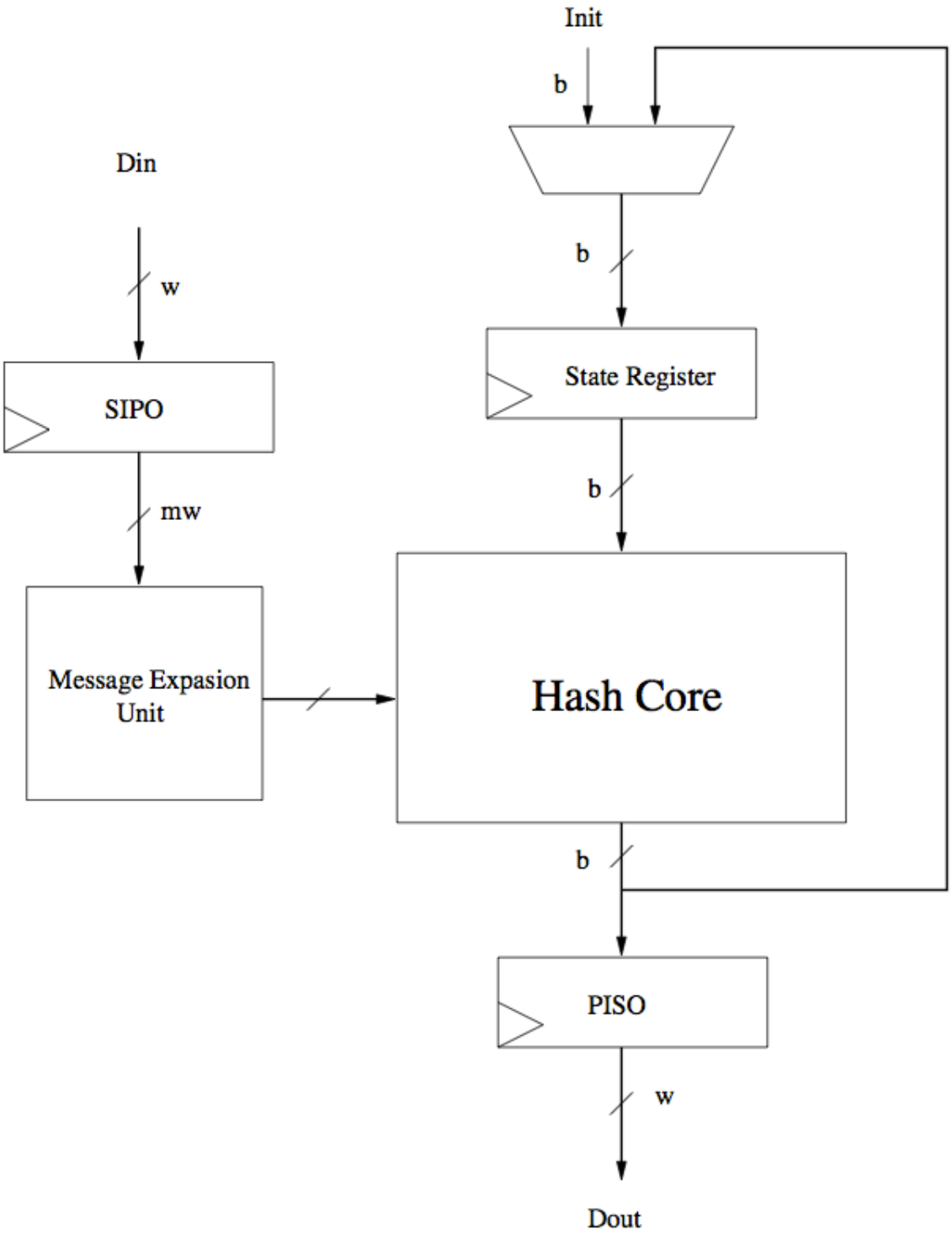


Figure 3.2: Generalized Datapath [13]

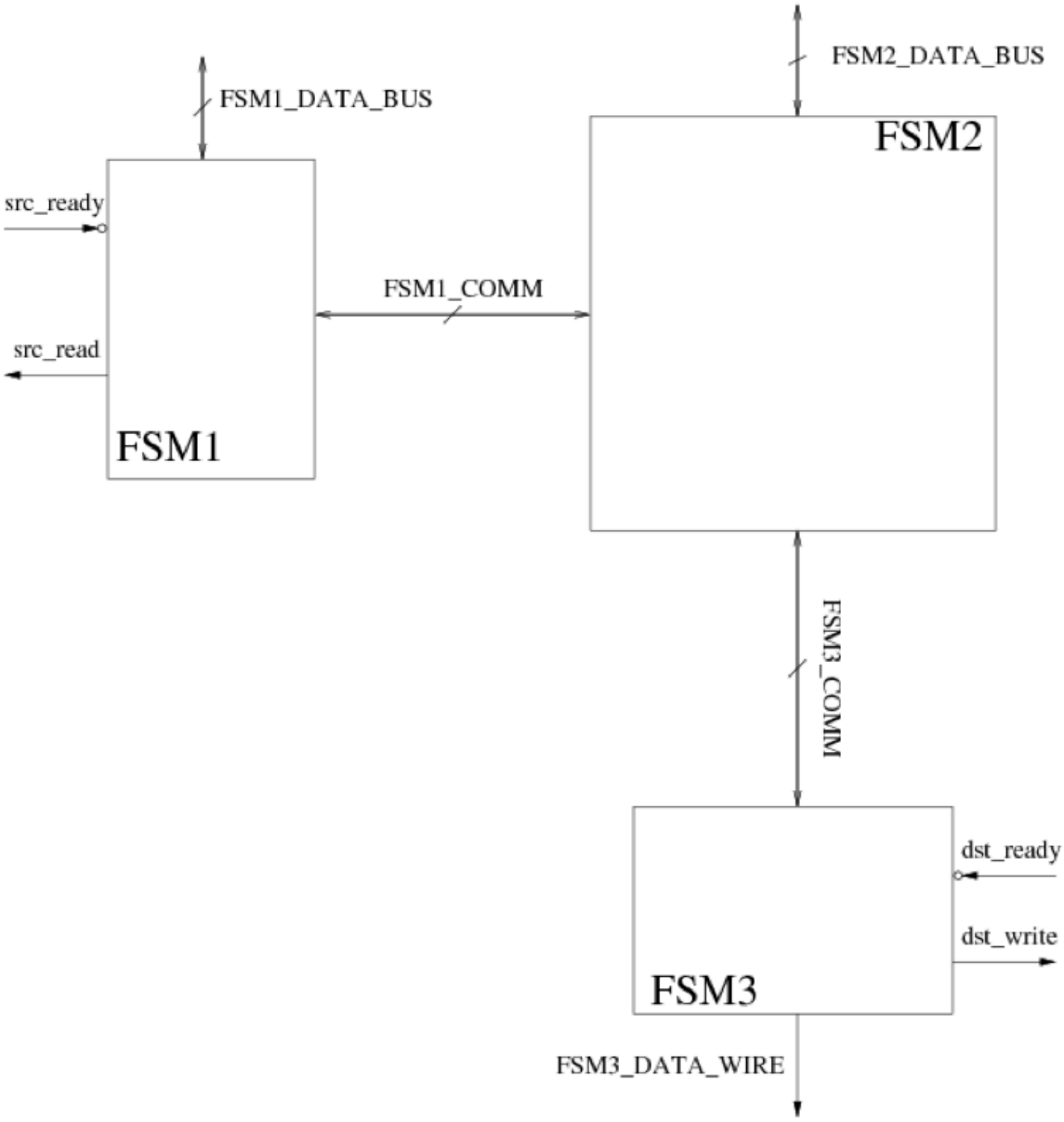


Figure 3.3: Generalized Controller [13]

3.2. Uniform Interface

A uniform interface has been used for all the designs. This interface was previously designed for the George Mason University SHA-3 research and is meant to be as simple as possible and require the fewest pins possible. There is a clock and reset, incoming and outgoing data buses that are each *w* pins wide (with *w* defined as the I/O Data Bus Width, 64 bits for BMW and CubeHash both), and pins for fifo operation and status fifoin_full, fifoin_write, fifoout_empty, and fifoout_read. There is sometimes an io_clk pin required due to clocking the FIFO and SIPO units faster than the main hash core. The total pin requirement for a design is given by the equation (3.1).

$$Pins = 6 + 2w + hasIoClk \qquad (3.1)$$

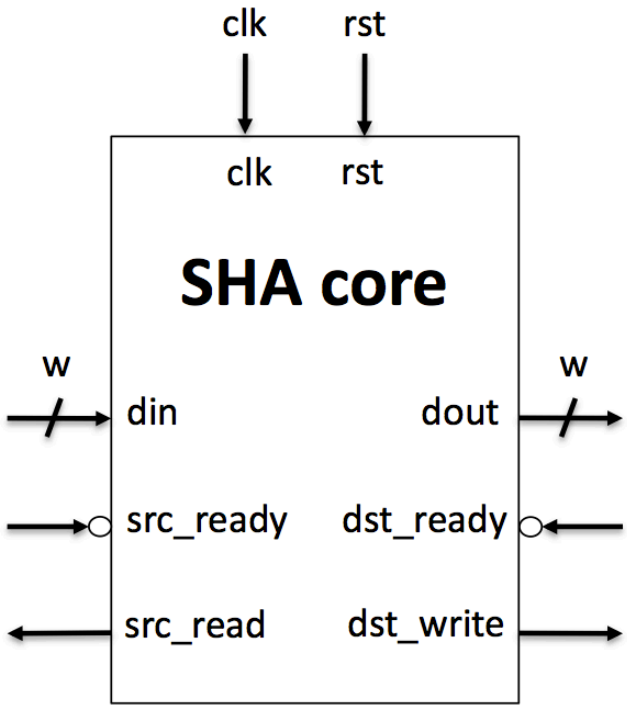


Figure 3.4: Interface to the SHA Core [13]

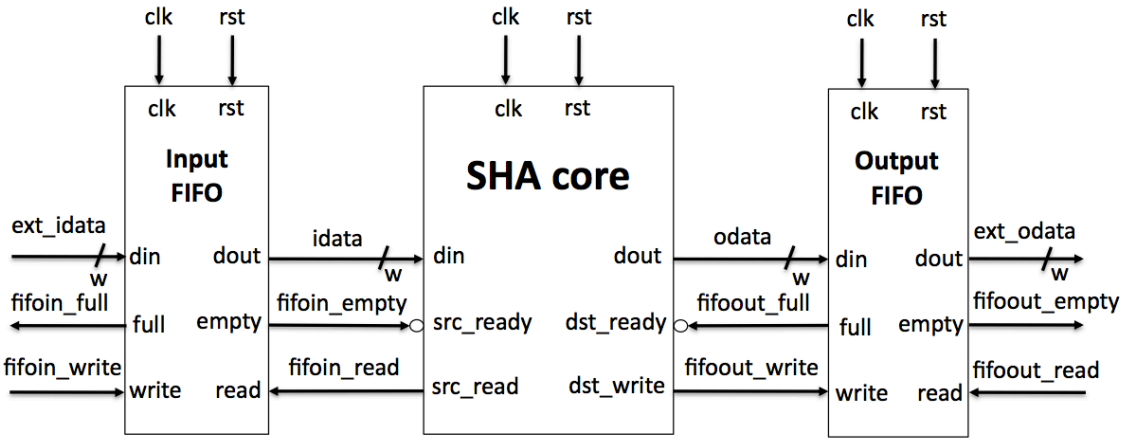


Figure 3.5: Full Interfaces for Typical Application [13]

3.3. Data Protocol

The George Mason University Cryptographic Engineering Research Group designed the data protocol used for the experiments done in this paper. The latest version of this protocol was used, correcting minor bugs present in earlier versions. This protocol is fully compatible with the Round 2 algorithms being used in this study and allowed for different lengths of input messages to be used as test vectors. Additionally, the protocol allows for a Message ID word to be sent when using a multi-message architecture, or omitted for a single-message architecture.

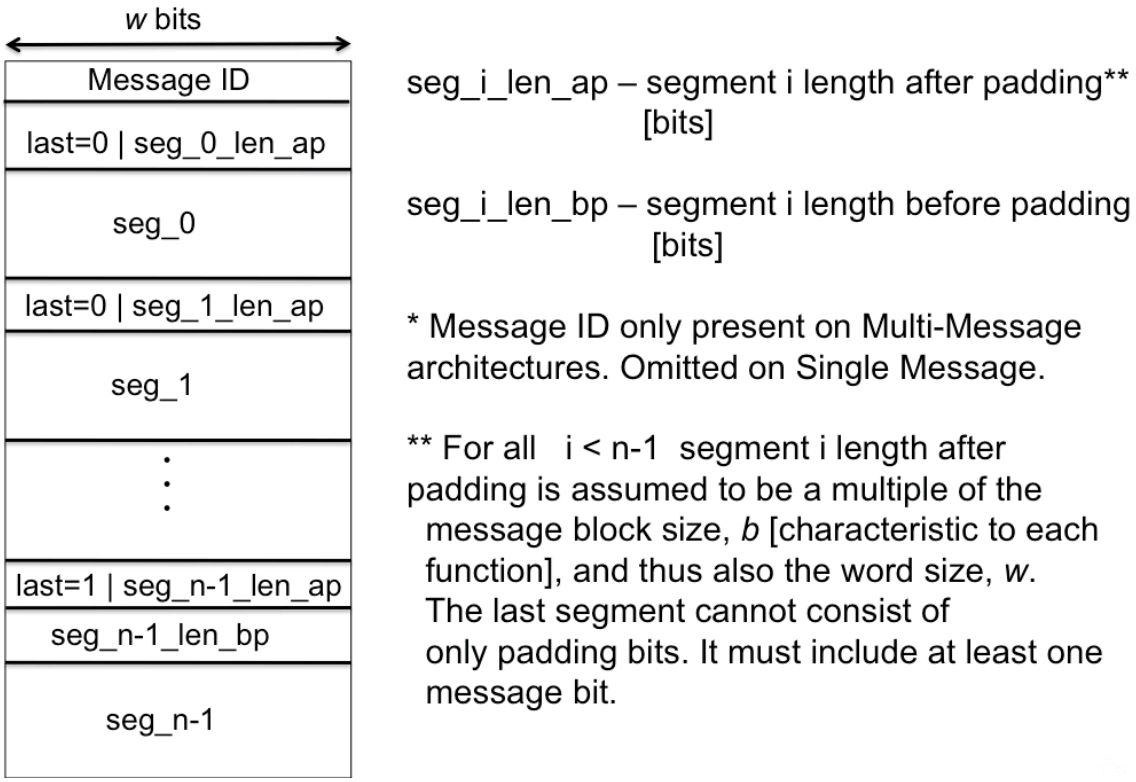


Figure 3.6: Data Protocol for SHA-3 Designs

CHAPTER 4: Basic Iterative Designs

The Basic Iterative designs, denoted as x1, are meant to be the basic average architectures of the given algorithms. Creating balanced architectures are of interest because in many cases a designer may wish to create hardware that has medium performance and takes a medium amount of chip area.

4.1. Blue Midnight Wish Basic Iterative Design

Blue Midnight Wish, in the Basic Iterative design, takes one clock cycle per round. The round consists of functions *F0, F1, F2*. This is a fully unrolled design for the algorithm and is what the specification for this hash function describes. The only alternative to a fully unrolled architecture is to perform horizontal folding of the internal functions. The basic iterative design does not do this because the folding significantly decreases throughput and also decreases area significantly. A designer looking for an "average" architecture would be more likely to choose this fully unrolled design.

There are several "rewiring" type transformations where bits are reorganized, and there are several data dependencies in the algorithm. Specifically, F1 has a very significant "triangular" data dependency as described in the cryptanalysis of the algorithm. While gaining significant cryptographic strength from these attributes, it also leads to a resulting hardware implementation that is complicated and congested. The

implementation tools from both vendors used in this study had significant problems with Blue Midnight Wish in its basic iterative architecture, specifically the 512 bit message digest version. Other than being time consuming and frustrating to work with when designing, this has another more serious impact in that the tools often were not able to complete placement and routing of the design on a device that should have been able to hold the design. For example, choosing an FPGA device that should only have 80% of its area utilized by Blue Midnight Wish 512-bit would very often fail to complete placement and routing. The tools would try for 10 to 14 hours and often still give up due to the extremely dense and congested design. The impact of this problem is that a designer may have to buy a larger and much more expensive device to hold the algorithm.

Since Blue Midnight Wish can potentially complete a hash in two clock cycles (one round, one finalization round), the number of clock cycles required to hash a message could be less than the number of clock cycles required to read that message in to the SIPO and write it out of the PISO. Therefore, a faster clock called *io_clk* is used to drive the input and output circuitry. On the 256-bit message digest version this is 8 times the regular clock speed, and on the 512-bit message digest version this is 16 times the regular clock speed. This ratio is determined by dividing *block_size / word_size*.

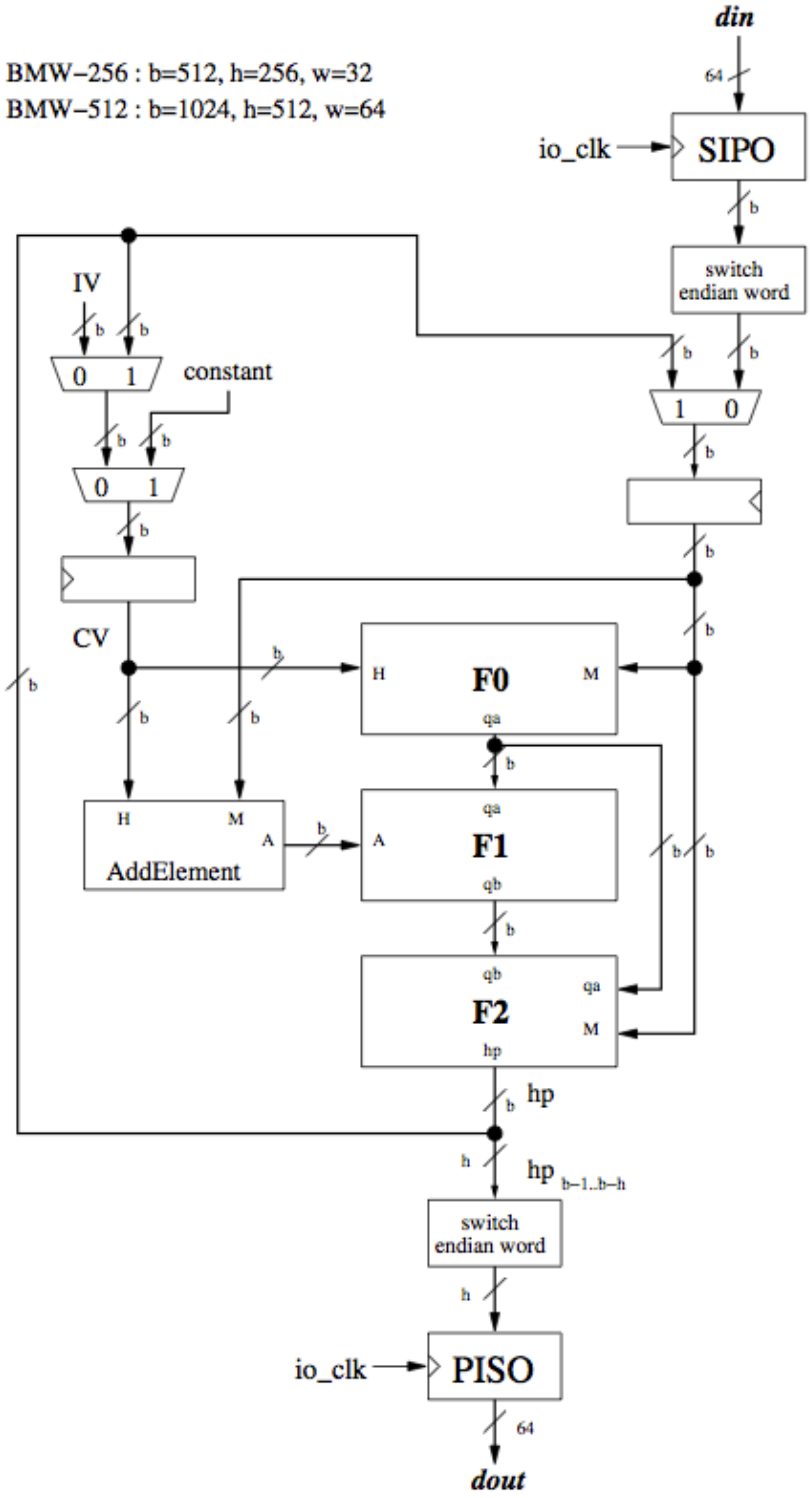


Figure 4.1: Blue Midnight Wish Basic Iterative Datapath [11]

The throughput for the Blue Midnight Wish Basic Iterative design is purely dependent on the clock speed that can be obtained for the unrolled logic. The latency is slightly impacted by set-up and finalization clock cycles, but is significantly impacted by the number of blocks in the message that must be processed. In the notation for latency, the clocks are shown to be */8* on 256-bit message digest version and */16* on the 512-bit message digest version. This is due to the *io_clk* that runs at this higher rate, and is shown for clarity.

Table 4.1: Latency and Throughput Equations for Blue Midnight Wish x1

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
BMW x1	2+8/8+N+1	512/T	2+**16/16**+N+**8/16**	**1024**/T

4.2. CubeHash Basic Iterative Design

The Basic Iterative version of the CubeHash algorithm was implemented in a manner consistent with the specification that allows for a variable number of rounds. Since the rounds can vary within a wide range, the implementation is of one of these rounds. The surrounding datapath and incoming control logic is responsible for bringing the message through the round the correct number of times. When looking at the results, it's clear that this architecture does not max out at the top frequency possible for the FPGA devices, so there is no physical implementation reason that the basic iterative design should included more than one round within a single clock. If the maximum frequency

of the FPGA devices was being reached with this design though, perhaps in future device families, a small architecture like this could be more efficient in terms of throughput by unrolling further.

Although the round of CubeHash is simple, there is one nuance to the algorithm that should be noted. When examining the round, it may appear that it would be slow due to large 512-bit operand adder units. However, all operations within the round are performed on 32-bit words of the data bus. Therefore, a 512-bit operand adder is actually implemented as 16 32-bit operand adders with the carry bits being discarded. Due to this property, the hardware implementation of the round is quite fast due to a short critical path. This attribute apparently does not lead to any cryptographic weakness, while keeping the throughput of hardware implementations as high performance as possible.

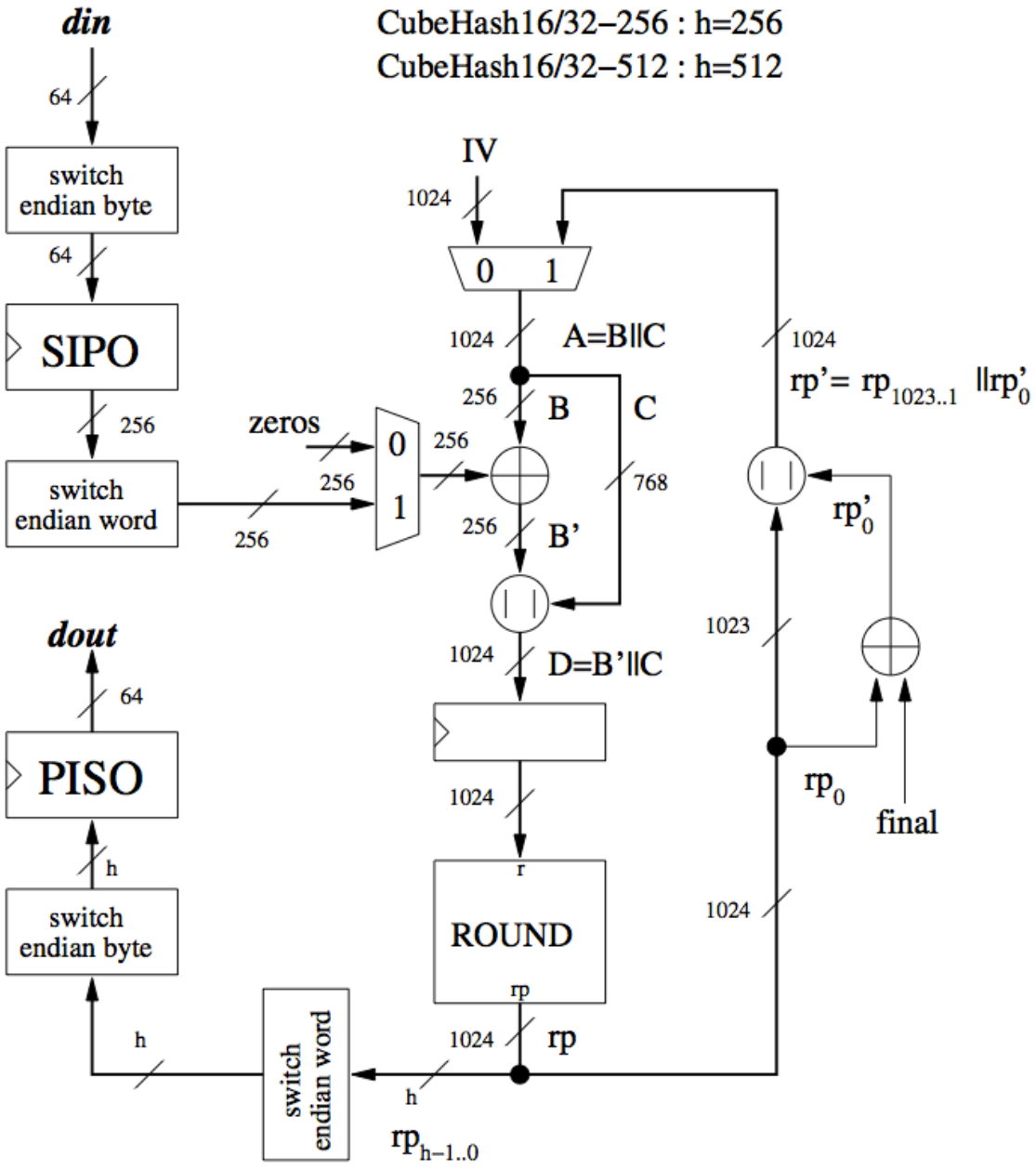


Figure 4.2: CubeHash Basic Iterative Datapath [11]

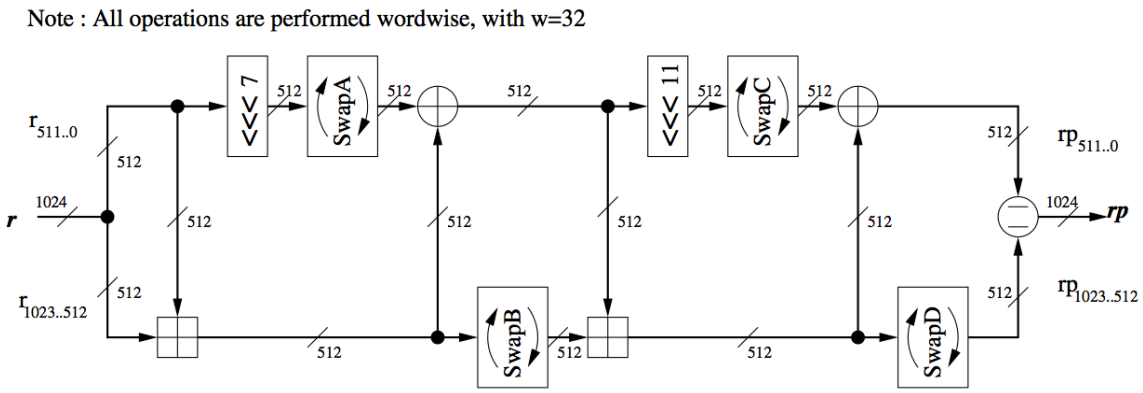


Figure 4.3: CubeHash Basic Iterative Round [11]

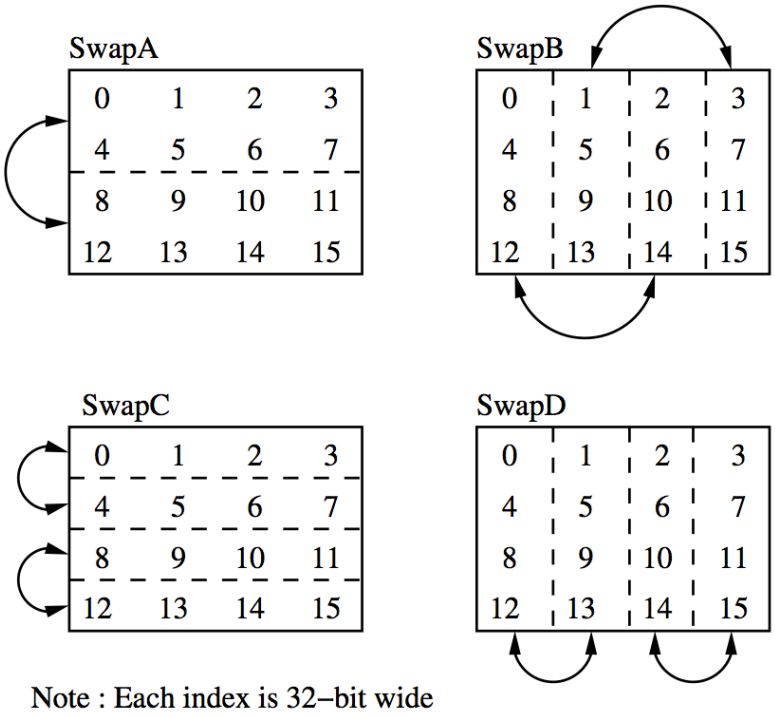


Figure 4.4: CubeHash Basic Iterative Round Swap Functions [11]

An interesting property of CubeHash is that internally the structure of the algorithm is not affected by the selected size for the message digest. The only impact of the message digest is at the end, when the output of the hash function is truncated to the message digest size. This is convenient in some ways, but is problematic in terms of throughput on the 512-bit version in comparison to most of the other SHA-3 candidates including Blue Midnight Wish. The reason is seen in Table 4.2, that the throughput of CubeHash is the same for both message digest versions since the message is read and run in the same manner for both. The latency of CubeHash has a large constant factor to it since finalization is 10 times the selected number of rounds, with that number being selected as 16 for our work. Therefore a very short message would incur a relatively high latency cost as compared to other algorithms that wouldn't have such a lengthy finalization.

Table 4.2: Latency and Throughput Equations for CubeHash x1

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
CubeHash x1	2+4+16*N+160+4	256/(16 * T)	2+4+16*N+160+**8**	256/(16 * T)

CHAPTER 5: Folded Designs

Folded designs are suitable for situations where area is a constrained resource and the throughput of the design is not of high importance. In these applications, the very low performance of a true low-area microprocessor-type design may not be acceptable. An example of a potential application could be a System on Chip that requires the ability to process SHA-3 hash messages, but where this is not the primary capability of the system. In general, these designs are of interest for SHA-3 because most nodes of a network or computing system computing hash message digests are doing so on a very infrequent basis and would want a low area and inexpensive solution. An exception to this would be high traffic network hardware that processes hash values.

5.1. Blue Midnight Wish Folded Design

The /32(h) folded Blue Midnight Wish design was created with the intention of reducing total area at the expense of throughput. Re-using portions of the circuit reduces the total area of the design. The x1 basic iterative design has a fully unrolled chain of all operations, and this /32(h) design runs only $1/32^{nd}$ of the operations per clock. Therefore, the clock frequency will be significantly increased because the critical path shrinks significantly. The hope is that the frequency of the circuit will increase enough to mitigate the fact that the clock must now run 32 cycles to complete a block of the hash.

In the Blue Midnight Wish x1 design, F0 takes the entire message at once, breaks it in to 16 parts, and has dedicated circuits to compute each of those 16 parts at once. The /32(h) design uses one of those circuits, modifies it to be generic enough, and uses that one circuit sixteen times sequentially. This generic circuit is slightly slower than having dedicated hardware for each of the 16 parts, due to the large muxes. However, the resulting area is lower.

In the Blue Midnight Wish x1 design, F1 accepts the entire message at once. The /32(h) design instead accepts 16 blocks one at a time after F0 finishes with them. After the 16 rounds of F0, F1 begins and runs an additional 16 times. The design starts computing on the 14th round of F0, instead of the 16th. This approach works around some data dependence issues and allows the large 17-operand addition (bottleneck) to occur more efficiently over 2 rounds, by grabbing the inputs to the equation early. Arrangement of the critical path was done such that operations were placed between folding stages as efficiently as possible.

No changes were done to F2 portion. It runs combinational logic during the last cycle and there were no obvious optimizations to area that could be done to re-use hardware components in a reasonable manner.

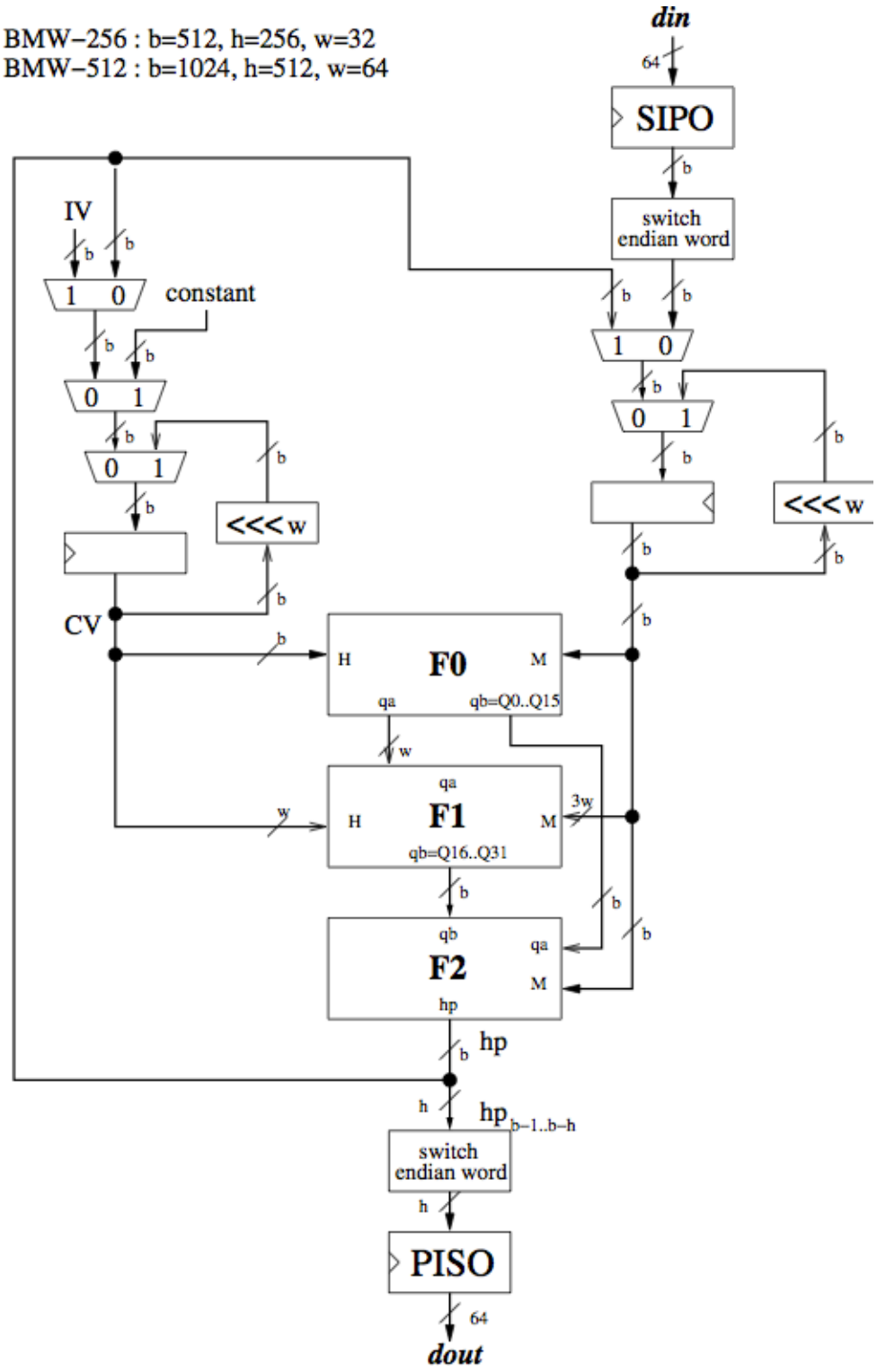
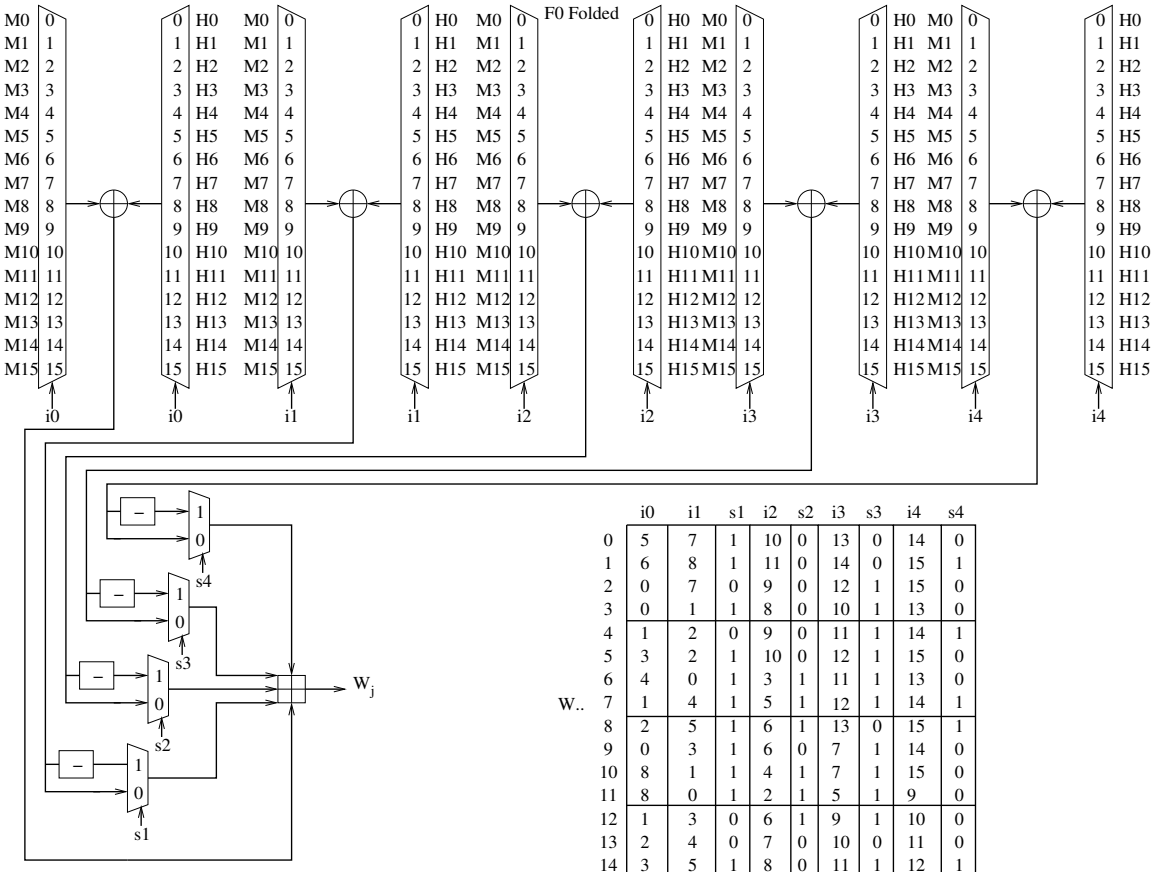


Figure 5.1: Blue Midnight Wish /32(h) Folded Datapath



W..	i0	i1	s1	i2	s2	i3	s3	i4	s4
0	5	7	1	10	0	13	0	14	0
1	6	8	1	11	0	14	0	15	1
2	0	7	0	9	0	12	1	15	0
3	0	1	1	8	0	10	1	13	0
4	1	2	0	9	0	11	1	14	1
5	3	2	1	10	0	12	1	15	0
6	4	0	1	3	1	11	1	13	0
7	1	4	1	5	1	12	1	14	1
8	2	5	1	6	1	13	0	15	1
9	0	3	1	6	0	7	1	14	0
10	8	1	1	4	1	7	1	15	0
11	8	0	1	2	1	5	1	9	0
12	1	3	0	6	1	9	1	10	0
13	2	4	0	7	0	10	0	11	0
14	3	5	1	8	0	11	1	12	1
15	12	4	1	6	1	9	1	13	0

Figure 5.2: Blue Midnight Wish /32(h) Folded F0 Function Part 1

F0 Folded, Part 2

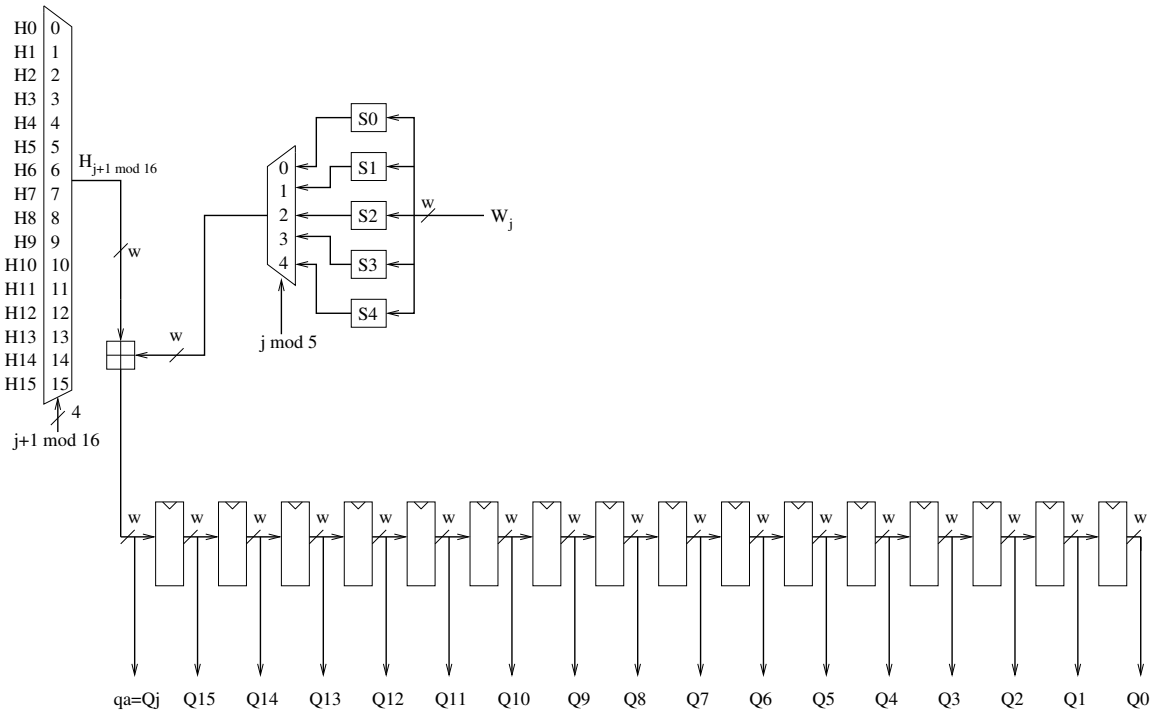


Figure 5.3: Blue Midnight Wish /32(h) Folded F0 Function Part 2

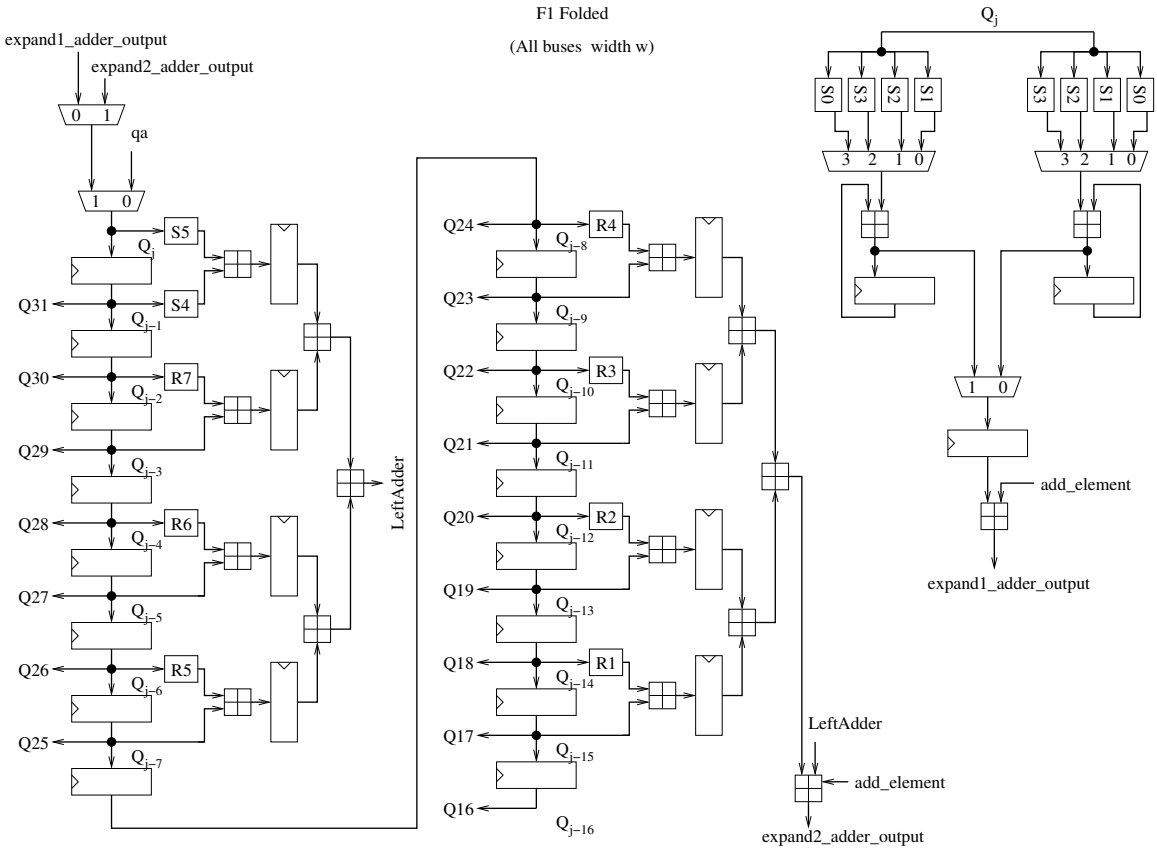


Figure 5.4: Blue Midnight Wish /32(h) Folded F1 Function

The latency and throughput of the folded /32(h) architecture differs from the basic iterative x1 architecture only because the number of rounds is increased to 32, so a multiplier of 32 has been added to the equations. For throughput, this means that the clock frequency would need to increase by 32 times to obtain the same throughput of x1. Similarly, latency will become worse if the clock frequency does not increase by 32 times. Since it's unlikely that the clock frequency gain will be this high, the expected result is to see a decrease in throughput and higher latency both at factor related to the ratio of the /32(h) clock frequency compared to the x1 clock frequency.

Table 5.1: Latency and Throughput Equations for Blue Midnight Wish /32(h)

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
BMW /32(h)	2+8/8+32*N+1	512/(T*32)	2+**16/16**+32*N+**8/16**	1024/(T*32)

5.2. CubeHash Folded Design

The folded design chosen for CubeHash was a conservative /2(h) horizontally folded design of the algorithm. CubeHash instead shortens the round by a factor of two while using muxes to select between some functionality, and completes this folded round twice as many times. In the case of the /2(h) architecture, the rotations and swaps are selected using muxes, but the adder and xor operations are always used. When compared to x1, it can be seen that since all the swap and rotation logic remains in place, the savings comes from only requiring one adder and one xor operation. Two clock cycles through this round are required to complete one round as defined in the specification.

Overall, this folded architecture saves space on one set 16 32-bit adders, and on one set of 16 32-bit adders, at the expense of two 512-bit muxes and very minimal control logic overhead. The expected impact on area is modest, but is important to explore to determine just how much area can be saved.

There are obviously additional folded architectures that are not explored as part of this paper, because they were considered too special purpose such that they are really in the realm of true low-area application. One of these approaches would be to horizontally fold the adder and xor logic. The addition must start computing first, but as each 32-bit

word of the addition is complete it can be sent to the xor. After 33 clock cycles, the first addition and xor would be complete and the data would be ready for the second half of the round following swap operations, at which point another 33 clock cycles would be required for the round. This would almost certainly make the circuit implement on hardware at the maximum frequency but would also require approximately 66 clock cycles per round, although there could be optimization to this with clever analysis of the swap functions. Even with optimization however, this architecture would clearly have a severe impact on throughput and latency. This impact was severe enough that the horizontally folded architecture was not investigated in this research. Another approach that would definitely be too special purpose, but likely exists in a practical form, is an instruction-set architecture where only one 32-bit adder and one 32-bit xor exist, along with swap and rotate logic, with control logic reading instructions from RAM and performing a long series of operations.

Note : All operations are performed wordwise, with w=32

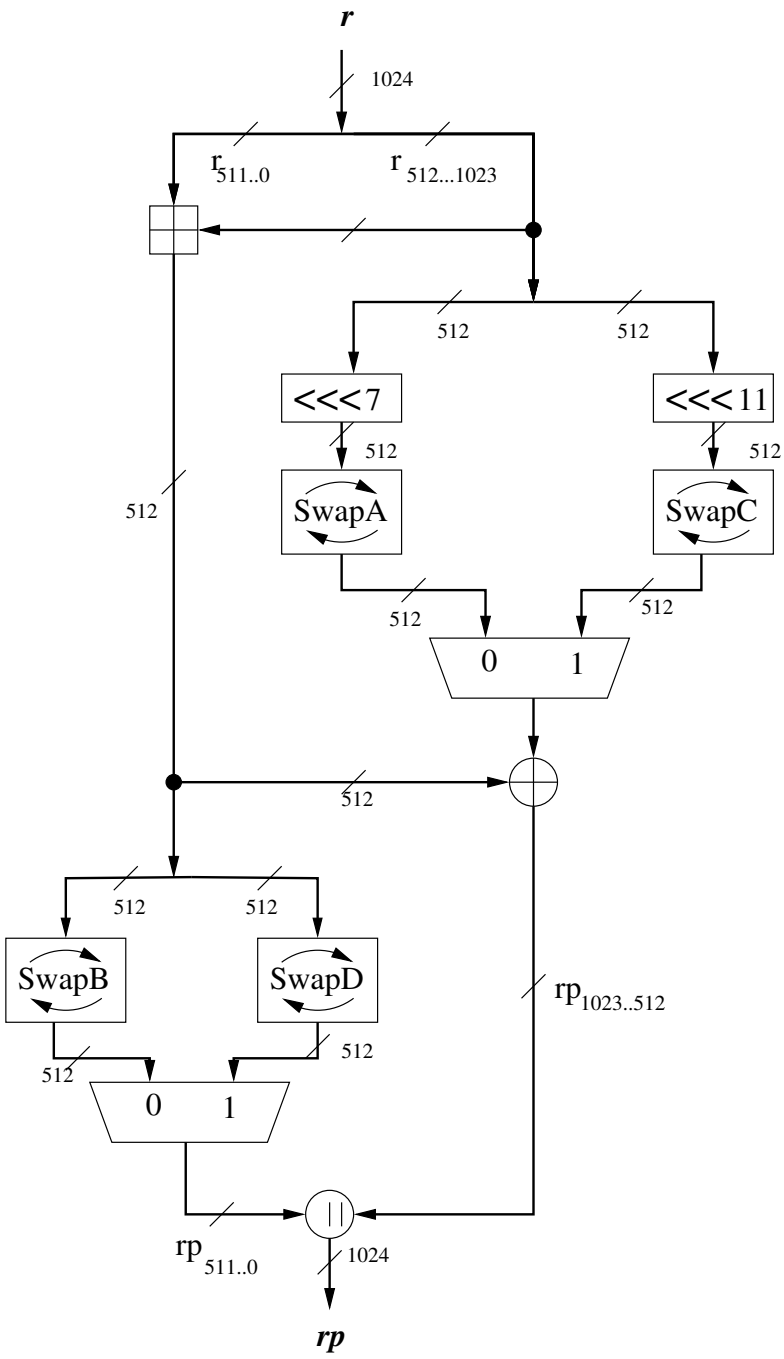


Figure 5.5: CubeHash Round Folded /2(h)

The impact to throughput and latency with this /2(h) folded architecture is that the previously existing constant multipliers of 16 are doubled to 32. In terms of throughput this means that the throughput would be halved compared to the Basic Iterative x1 architecture at the same clock frequency. The latency suffers similarly with the rounds each taking twice as long, to include the 10 sets of 16 rounds for finalization also taking twice as long.

Table 5.2: Latency and Throughput Equations for CubeHash /2(h)

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
CubeHash /2(h)	2+4+32*N+320+4	256/(32 * T)	2+4+32*N+320+**8**	256/(32 * T)

CHAPTER 6: Multi-Message Designs

Multi-message designs have the potential to be the highest possible throughput architecture for a given algorithm. Blue Midnight Wish is a large design and much of the datapath is "idle" in the sense that it computes a result and waits a long time for the next clock cycle before doing new work. This makes it an ideal candidate for pipelining, because that long unrolled idle datapath can be made to work much more efficiently. CubeHash does not have much opportunity for pipelining, but use its small size to its advantage by having multiple identical cores in parallel on the device all processing message digests of different messages simultaneously. Since the entire datapath of CubeHash is generally not waiting, this technique leads to high throughput and efficient use of area.

6.1. Blue Midnight Wish Pipelined Design

In the x1-PPL18 architecture of Blue Midnight Wish, placing registers at the end of the F0 function creates the first of the 18 pipeline stages. Next, the 16 expansion functions of the F1 function all of their outputs registered to create 16 pipeline stages. Finally, F2 has its output registered to make the 18^{th} and final pipeline stage of the design.

An alternative architecture was designed for this work that split each expand function in to two pipelined stages. The design of this architecture had the advantage of

splitting up the addition operations in an even manner. Instead of a longer 17 operand addition, the first pipeline stage of an expand function computed the sum of 9 operands, and the second stage summed that result along with the other 8 operands. One of the operands passed to the second stage was *add_element*, so that the work in computing that value was not in the critical path. Interestingly, there was almost no improvement to the critical path and clock frequency of this experimental x1-PPL34 design. The only impact of moving to this design was to make latency worse, keep throughput basically the same, and to require almost twice as many flip flops. Therefore, this x1-PPL34 design was abandoned as inferior.

Note that not only the quadruple pipe *Q* is brought through each pipeline stage, but also the *M* and *H* as well except that H is not in the final stage. Registering this much data is why this design has such a huge utilization of flip flops.

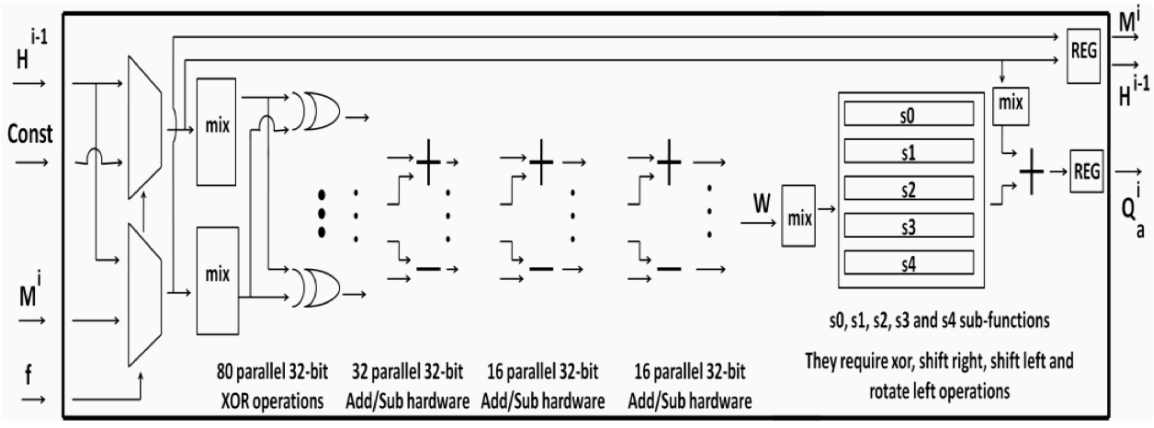


Figure 6.1: Blue Midnight Wish Pipelined F0 Function [8]

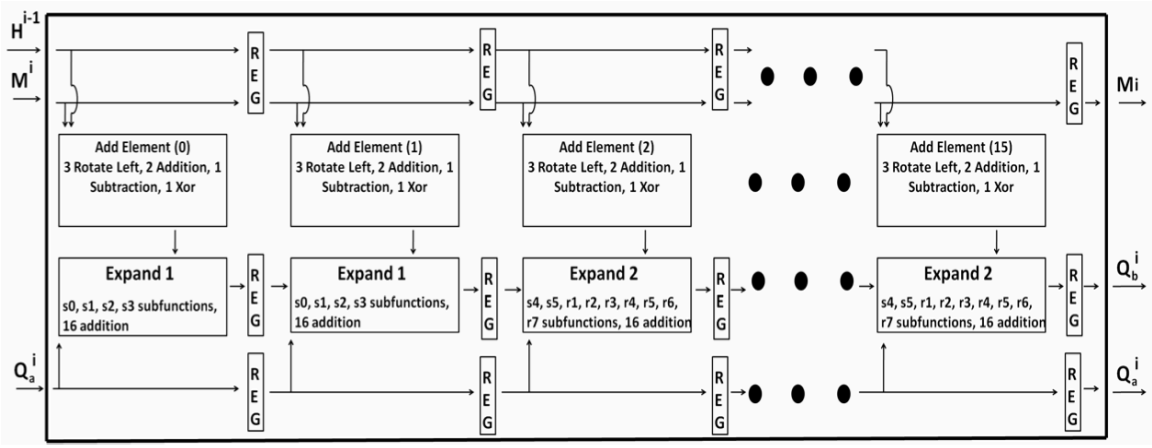


Figure 6.2: Blue Midnight Wish Pipelined F1 Function [8]

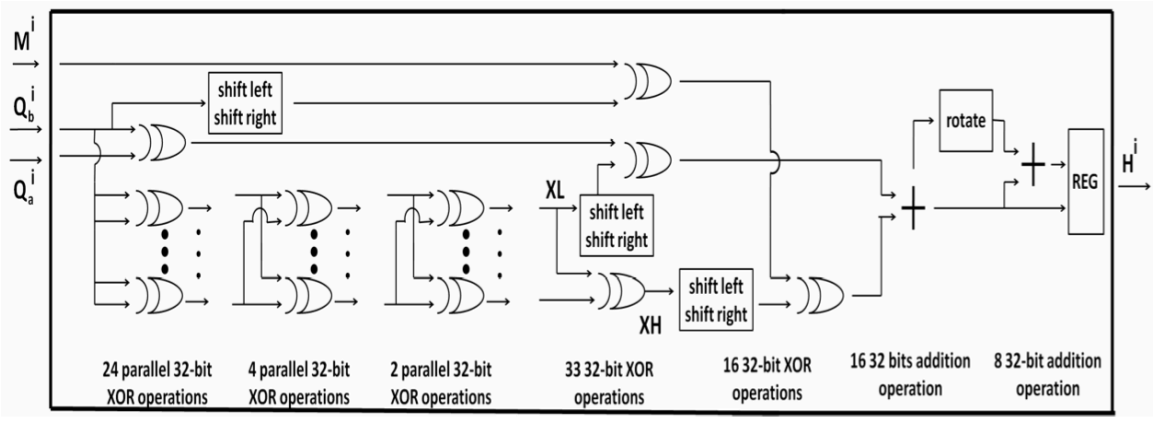


Figure 6.3: Blue Midnight Wish Pipelined F2 Function [8]

The significant problems with physical implementation that were encountered on the Basic Iterative version were mitigated significantly with this architecture due to the presence of registered pipelined stages. This results in much less congestion in the design and allows the tools to more successfully place and route the components. As discussed, an issue encountered with the pipelined design was that there was not enough

flip flops available on devices even though there were plenty of leftover Slices or ALUTs because so many registers were required to hold intermediate results between the 18 pipeline stages. This is interesting because in general, registers are considered "free" in FPGAs because they are so abundant. However, the registers do become the limiting factor, and a larger more expensive device must be used to hold the implemented architecture. This indicates that in ASIC implementation, where registers require significant surface area, the x1-PPL18 architecture would be extremely large. In fact, Savas et al did find that a x1-PPL18 version of Blue Midnight Wish on ASIC consume almost three times the area of the Basic Iterative x1 architecture.

The impact of pipelining on latency is that *n* clock periods must complete for a round to finish, with *n* from x1-PPL*n*. Therefore, similarly to folding, whether latency improves or worsens depends on how much the clock frequency is increased by breaking the critical path in to shorter sections. The number of pipeline stages multiplies the throughput, since that many messages are being concurrently processed. However, that same factor of pipeline stages multiplies the denominator as well because it now takes *n* clock cycles to complete processing a block of the message.

Table 6.1: Latency and Throughput Equations for Blue Midnight Wish x1-PPL18

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
BMW x1-PPL18	2+8/8+N*18+1	(18*512)/(T*18)	2+**16/16**+N*18+**8/16**	(18*1024)/(T*18)

6.2. CubeHash Multiunit Design

The CubeHash Multiunit designs were created as parallel units operating on the same physical device. Having multiple parallel cores is expected to give a linear increase in throughput compared to the area spent, with granularity being available in steps as small as the area of one unit. As more parallel units are implemented on a device, there should not be any meaningful impact on clock speed or area of the individual units. There is slight overhead to the FSM3 control and operation, but in general this is negligible.

As described in the general technique for creating a Multiunit design, the most suitable type of algorithm for a Multiunit design would be a lower area design. This makes the low area CubeHash a natural choice because it is one of the smallest SHA-3 Round 2 candidates.

There were 5 parallel architectures created with CubeHash. First, an x1-PAR5 design was created as a sort of baseline. This allowed initial benchmark measurement of the architecture, which was analyzed to get an approximate calculation for area per unit. Since a goal of the experiment was to compare CubeHash Multiunit architectures with Blue Midnight Wish pipelined architectures, the area of the corresponding Blue Midnight Wish x1-PPL18 design was divided by the approximate calculation for area per CubeHash unit. This calculation of *BMW x1-PPL18 Area / One CubeHash Unit Area* gives the number *n* for CubeHash x1-PAR*n* that should implement to roughly the same area as the Blue Midnight Wish *x1-PPL18* design. Interestingly, this resulted in four additional CubeHash architectures. When working with 256-bit message digest and 512-bit message digest sizes, the ratio was different. Additionally, when working in Xilinx

tools and in Altera tools, the ratio was different. This intuitively makes sense because the area of the discussed architectures was very different between these vendors and message digest sizes.

The latency of a Multiunit design is calculated exactly as in the Basic Iterative x1 architectures, since there is no change to the actual operation of each independent core. The throughput of a Multiunit design differs from the Basic Iterative x1 architecture because the number of parallel cores implemented in the design multiplies the throughput. Therefore, the throughput of *x1-PARn* is *n* times the Basic Iterative *x1* throughput.

Table 6.2: Latency and Throughput Equations for CubeHash x1-PAR*n*

	256-Bit		512-Bit	
	Hash Time [Cycles]	Throughput [Mbit/s]	Hash Time [Cycles]	Throughput [Mbit/s]
CubeHash x1-PAR5	2+4+16*N+160+4	(256/(16 * T)) * 5	2+4+16*N+160+**8**	(256/(16 * T)) * 5
CubeHash x1-PAR9	2+4+16*N+160+4	(256/(16 * T)) * 9	2+4+16*N+160+**8**	(256/(16 * T)) * 9
CubeHash x1-PAR17	2+4+16*N+160+4	(256/(16 * T)) * 17	2+4+16*N+160+**8**	(256/(16 * T)) * 17
CubeHash x1-PAR25	2+4+16*N+160+4	(256/(16 * T)) * 25	2+4+16*N+160+**8**	(256/(16 * T)) * 25
CubeHash x1-PAR33	2+4+16*N+160+4	(256/(16 * T)) * 33	2+4+16*N+160+**8**	(256/(16 * T)) * 33

CHAPTER 7: Results

Results were gathered using the ATHENa tool as described in Chapter 2: Design Methodology. The gathered data for the designs is the maximum clock frequency of the implemented design, and the area in terms of CLB Slices for Xilinx or ALUTs on Altera tools. Some of the CubeHash x1-PAR*n* architectures only have results for some vendors and message digest sizes, because only the x1-PAR5 and the closest match to the corresponding Blue Midnight Wish x1-PPL18's area were implemented.

Table 7.1: Results of All Implementations on Xilinx Virtex 5 Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [CLB Slices]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1	4.89	8.14	5004	4168	12039	6164	0.42	0.68
BMW x1-PPL18	46.51	70.97	47628	36335	24564	11610	1.84	2.96
BMW /32(h)	44.27	69.41	1416	1110	4001	2211	0.35	0.50
CH x1	152.70	151.93	2443	2430	745	672	3.28	3.62
CH x1-PAR5	151.26	151.56	12101	12125	3742	3490	3.23	3.47
CH x1-PAR17	N/A	150.49	N/A	40933	N/A	11360	N/A	1.91
CH x1-PAR33	150.65	N/A	79542	N/A	24570	N/A	3.24	N/A
CH /2(h)	173.52	184.71	1388	1477	733	624	1.89	2.37

Table 7.2: Results of All Implementations on Altera Stratix III Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [ALUTs]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1	9.11	10.04	9328	5140	24718	12332	0.38	0.42
BMW x1-PPL18	82.72	95.62	84705	48957	48037	17233	1.67	2.69
BMW /32(h)	58.30	73.82	1865	1181	11997	6057	0.16	0.20
CH x1	185.08	204.08	2961	3265	1923	1919	1.54	1.70
CH x1-PAR5	211.15	222.77	16892	17821	9533	9513	1.77	1.87
CH x1-PAR9	N/A	223.06	N/A	32078	N/A	17112	N/A	1.87
CH x1-PAR25	202.10	N/A	80840	N/A	47492	N/A	1.70	N/A
CH /2(h)	329.38	324.78	2635	1477	1803	1657	1.46	0.89

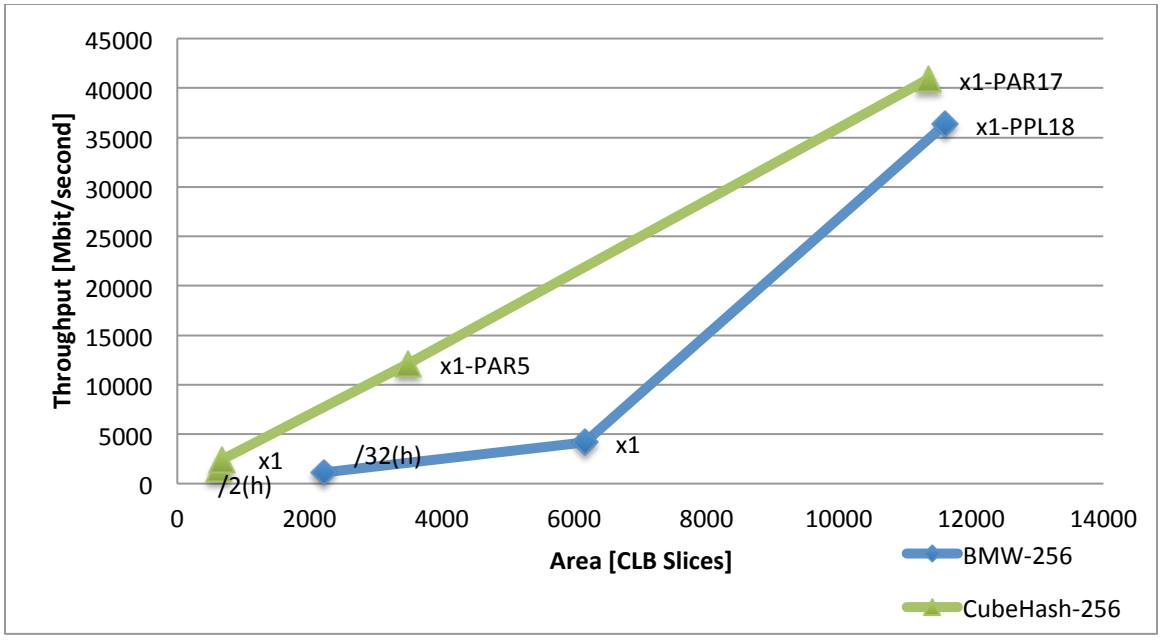


Figure 7.1: Xilinx Virtex 5 Family Results – 256 Bit Message Digest

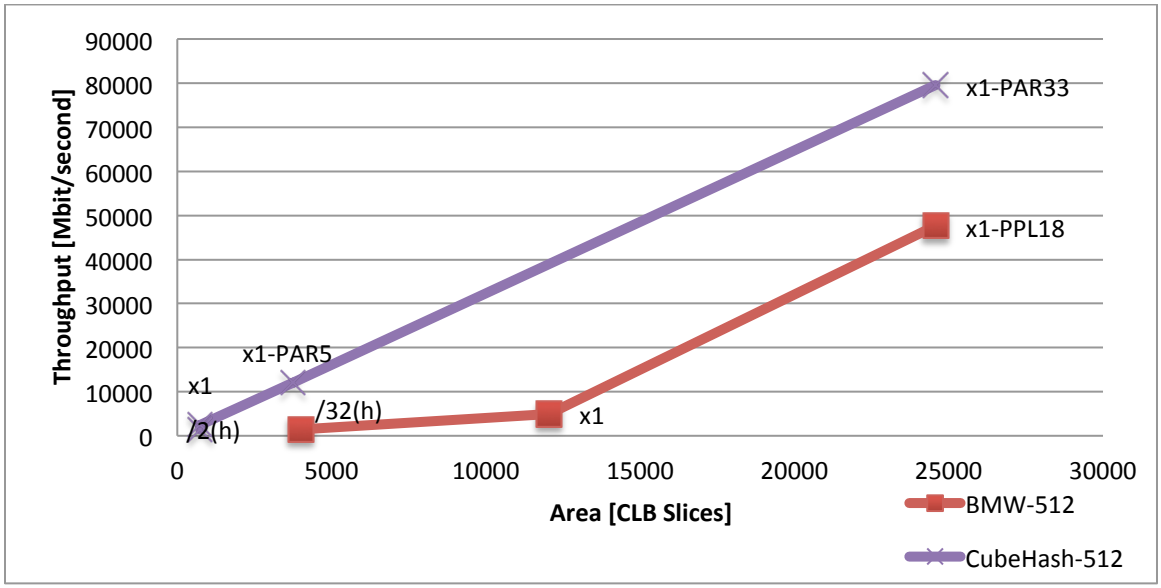


Figure 7.2: Xilinx Virtex 5 Family Results – 512 Bit Message Digest

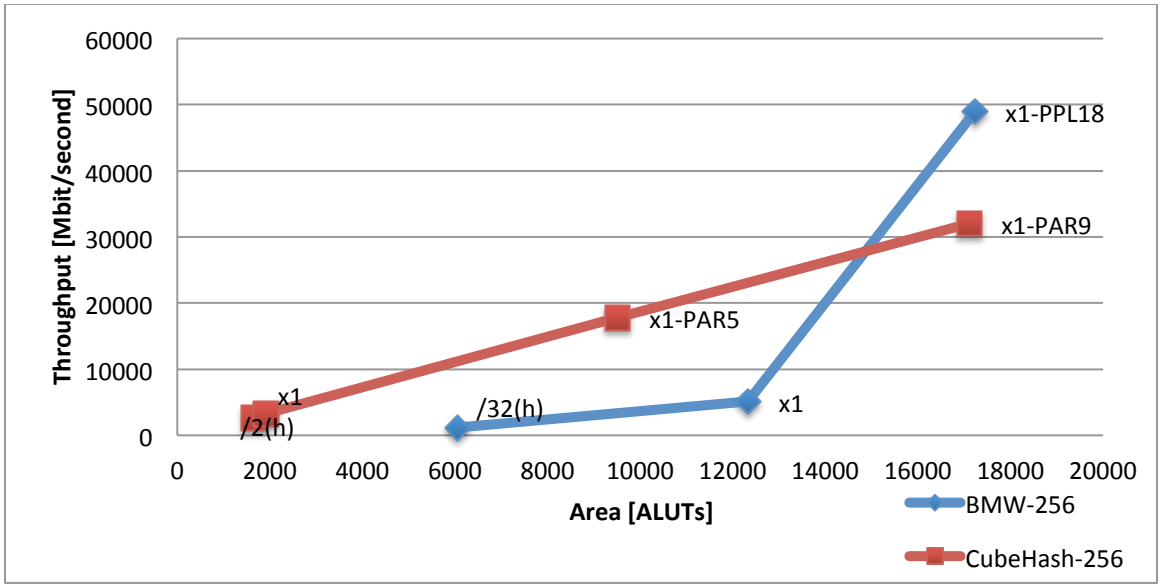


Figure 7.3: Altera Stratix III Family Results – 256 Bit Message Digest

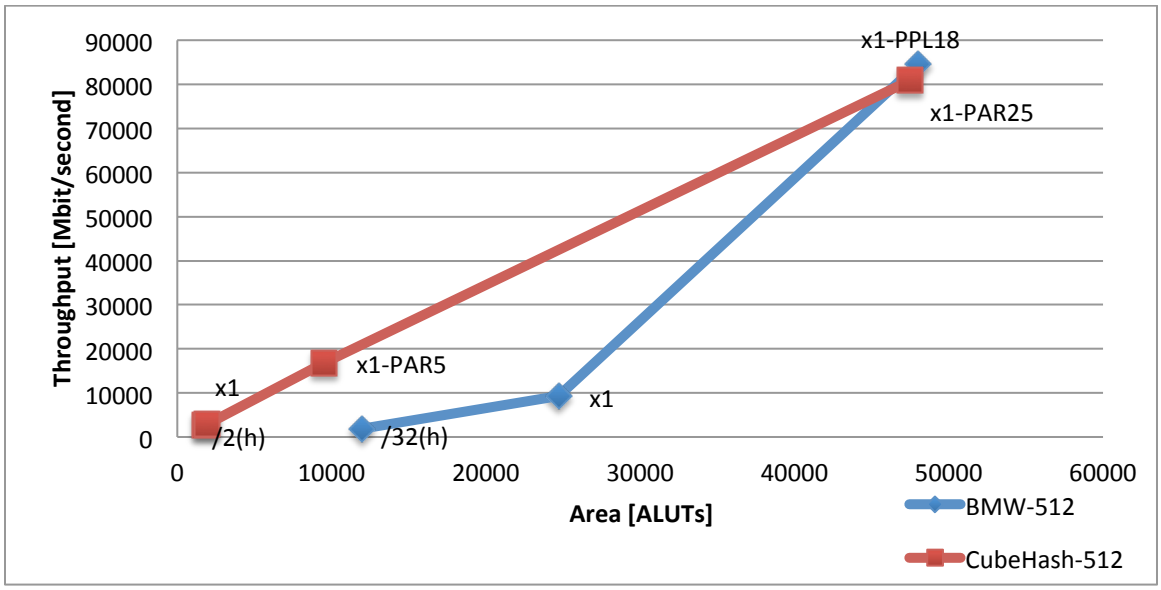


Figure 7.4: Altera Stratix III Family Results – 512 Bit Message Digest

7.1. Discussion of Basic Iterative Designs

The Basic Iterative results of Blue Midnight Wish and CubeHash tell a very clear story that CubeHash is much smaller in terms of area and Blue Midnight Wish attains much higher throughput. Examining the Throughput / Area calculations, it can be seen that CubeHash is significantly more efficient than Blue Midnight Wish. On Xilinx devices this is by a factor of roughly 6.27, and on Altera devices this is by a factor of roughly 4.05. The difference in these results can be quantified because when examining as a ratio, the Altera tools require more area for CubeHash. Also, the Altera tools give a significant boost to performance of Blue Midnight Wish 512-bit message digest version.

The Xilinx tools gave results that were expected, with the larger 512-bit message digest of Blue Midnight Wish requiring approximately twice the area and exhibiting roughly half the clock speed as compared to the 256-bit version. CubeHash 512-bit and 256-bit message digest versions behaved almost identically, with differences being easily attributed to random variation between place and route performed by the tools. There is a slight increase in the 512-bit message digest version's area because of additional surrounding and output logic.

In contrast, the Altera results are not what are intuitively expected. In terms of circuit area, the Altera results are similar to Xilinx and are what is expected. However, CubeHash's 512-bit version has a maximum clock frequency of 20MHz less than the 256-bit version, a roughly 10.7% variation. There is no reasonable explanation for this except that the results obtained were from particularly good or particularly bad placement and routes when implementing the design.

Table 7.3: Results of Basic Iterative Implementation on Xilinx Virtex 5 Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [CLB Slices]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1	4.89	8.14	5004	4168	12039	6164	0.42	0.68
CH x1	152.70	151.93	2443	2430	745	672	3.28	3.62

Table 7.4: Results of Basic Iterative Implementation on Altera Stratix III Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [ALUTs]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1	9.11	10.04	9328	5140	24718	12332	0.38	0.42
CH x1	185.08	204.08	2961	3265	1923	1919	1.54	1.70

7.2. Discussion of Folded Designs

Examining the results for the Folded Designs shows that Blue Midnight Wish gains much more in terms of area through folding than CubeHash does. For the Xilinx devices, the Blue Midnight Wish folded design uses 34.5% the area as the Basic Iterative design. On those same devices, the CubeHash folded design uses 95.5% the area as the Basic Iterative design. In terms of throughput, the folded Blue Midnight Wish attains only approximately 27.5% the throughput of the Basic Iterative design, and the folded CubeHash attains 59% of the throughput of the Basic Iterative design. The loss of throughput for Blue Midnight Wish is similar to what is expected for the gains in area, but for CubeHash the ratio is very poor since a significant amount of throughput is lost for very little gain in area.

Altera devices tell a slightly different story. Altera devices show Blue Midnight Wish using 49% of the area of the Basic Iterative design, and attaining approximately 21.5% of the throughput of the Basic Iterative design. This set of results shows a much more significant loss in throughput for Blue Midnight Wish, and a less significant ability to reduce the area. In contrast, CubeHash implements much better on the Altera devices, showing a utilization of 90% of the area of the Basic Iterative design, and a throughput that reaches approximately 84.5% of the Basic Iterative design. These results still show that CubeHash does not gain much area, but the gain in area is accompanied by a much more reasonable drop in throughput.

In terms of raw numbers instead of percentages of change, the folded architectures of the algorithms under both Xilinx and Altera have nearly identical throughput ability. On average, CubeHash does have a throughput advantage, but it is very close and on the Xilinx 512-bit message digest version Blue Midnight Wish does have a higher throughput than CubeHash. This close comparison of throughput shows that for the 1 to 2 Gbit/second throughput space, CubeHash requires roughly 4 to 6 times less area than Blue Midnight Wish.

There is very little granularity to the CubeHash /2(h) architecture. Folding more than twice is not very practical, and folding any less results in the Basic Iterative design. With the type of folded architecture used for Blue Midnight Wish /32(h) there is no reasonable way to further decrease area, but it may be possible to find a /16(h) architecture that performs with a higher throughput but does not gain as much in terms of area. This type of architecture may process two pieces of data at once. However, the

added complexity of this approach is likely to produce unattractive results for throughput as well as area.

Table 7.5: Results of Folded Implementations on Xilinx Virtex 5 Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [CLB Slices]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW /32(h)	44.27	69.41	1416	1110	4001	2211	0.35	0.50
CH /2(h)	173.52	184.71	1388	1477	733	624	1.89	2.37

Table 7.6: Results of Folded Implementations on Altera Stratix III Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [ALUTs]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW /32(h)	58.30	73.82	1865	1181	11997	6057	0.16	0.20
CH /2(h)	329.38	324.78	2635	1477	1803	1657	1.46	0.89

Table 7.7: Throughput and Area Ratio Comparison on Xilinx Virtex 5 Family for Folded Architectures

	Throughput [Ratio]		Area [Ratio]	
	512	**256**	**512**	**256**
BMW /32(h) compared to BMW x1	0.28	0.27	0.33	0.36
CH /2(h) compared to CH x1	0.57	0.61	0.98	0.93

Table 7.8: Throughput and Area Ratio Comparison on Altera Stratix III Family for Folded Architectures

	Throughput [Ratio]		Area [Ratio]	
	512	**256**	**512**	**256**
BMW /32(h) compared to BMW x1	0.20	0.23	0.49	0.49
CH /2(h) compared to CH x1	0.89	0.80	0.94	0.86

7.3. Discussion of Multi-Message Designs

The approach for the Multi-Message architectures was to develop and implement the Blue Midnight Wish x1-PPL18 architecture because it is not flexible in terms of how much area it uses – there are no design decisions with that Multi-Message architecture that can easily change the amount of area, so it is considered as a set factor. Then, the CubeHash Multiunit designs were tweaked using VHDL generics to create designs that used a very similar amount of area as the Blue Midnight Wish x1-PPL18 for that specific vendor's devices and message digest size. The results varied significantly depending on which vendor's devices and which message digest size were used.

For Xilinx devices on 256-bit message digest versions of the algorithms the CubeHash x1-PAR17 architecture performed modestly better, with approximately 13% better throughput than the Blue Midnight Wish x1-PPL18 architecture.

For Xilinx devices on 512-bit message digest versions of the algorithms the CubeHash x1-PAR33 architecture performed significantly better, with approximately 67% better throughput than the Blue Midnight Wish x1-PPL18 architecture.

For Altera devices on 256-bit message digest versions of the algorithms the Blue Midnight Wish x1-PPL18 architecture performed significantly better, with approximately 53% better throughput than the CubeHash x1-PAR9 architecture.

For Altera devices on 512-bit message digest versions of the algorithms the Blue Midnight Wish x1-PPL18 architecture performed slightly better, with approximately 5% better throughput than the CubeHash x1-PAR25 architecture.

Table 7.9: Results of Multi-Message Implementations on Xilinx Virtex 5 Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [CLB Slices]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1-PPL18	46.51	70.97	47628	36335	24564	11610	1.84	2.96
CH x1-PAR5	151.26	151.56	12101	12125	3742	3490	3.23	3.47
CH x1-PAR17	N/A	150.49	N/A	40933	N/A	11360	N/A	1.91
CH x1-PAR33	150.65	N/A	79542	N/A	24570	N/A	3.24	N/A

Table 7.10: Results of Multi-Message Implementations on Altera Stratix III Family

	Max Clock Freq [MHz]		Throughput [Mbit/s]		Area [ALUTs]		Throughput / Area	
	512	**256**	**512**	**256**	**512**	**256**	**512**	**256**
BMW x1-PPL18	82.72	95.62	84705	48957	48037	17233	1.67	2.69
CH x1-PAR5	211.15	222.77	16892	17821	9533	9513	1.77	1.87
CH x1-PAR9	N/A	223.06	N/A	32078	N/A	17112	N/A	1.87
CH x1-PAR25	202.10	N/A	80840	N/A	47492	N/A	1.70	N/A

CHAPTER 8: Conclusions and Future Work

In the Basic Iterative architecture, CubeHash shows a lot of flexibility because it can attain reasonably high throughput within a very small amount of area. Blue Midnight Wish gives better performance, but the high area requirement is likely unreasonable for many applications.

The folded architecture of Blue Midnight Wish showed a very significant reduction in the area used. This was a great flexibility and shows that the algorithm is very adaptable for different applications. The issue however is that, despite showing a huge improvement, the /32(h) Blue Midnight Wish design is still much larger than the x1 Basic Iterative CubeHash. This shows that, despite the large range of area that can be selected by a designer, Blue Midnight Wish is still at a very large disadvantage in terms of hardware area requirements as compared to CubeHash. CubeHash gains a very modest savings in area by folding in the case of the /2(h) architecture. This savings could still be important to a designer who is attempting to implement the smallest possible CubeHash without moving to a specialized true low-area implementation. In the folded designs, there was variation due to the vendor and devices used, but there was not significant variation and the results were quite clear to interpret.

The Multi-message versions of the algorithms showed a significant improvement in throughput for both Blue Midnight Wish and CubeHash. Selecting the same area for

the two algorithms on each experiment allowed a real comparison to be done between them without having to extrapolate results that may not reflect reality.

On Xilinx devices, CubeHash a higher throughput than Blue Midnight Wish on the 256-bit message digest version, and this throughput advantage is significantly improved when moving to the 512-bit message digest version of the algorithm. This large improvement is due to the inherent way that CubeHash works with message digests, since it uses the same area for lower throughput on the 256-bit message digest version.

On Altera devices, Blue Midnight Wish has a much higher throughput than CubeHash on the 256-bit message digest version, and Blue Midnight Wish has a modestly higher throughput than CubeHash on the 512-bit message digest version. Again, this difference between the measures of throughput can be attributed to the fact that CubeHash exhibits lower throughput to area ratio on the 256-bit message digest version.

Therefore, it can be seen that the Multi-message CubeHash architectures have a significant throughput on Xilinx devices when compared to Blue Midnight Wish, and the opposite is true on Altera devices. The advantage exhibited in favor of CubeHash on the Xilinx devices is higher than the advantage exhibited in favor of Blue Midnight Wish on the Altera devices, but only to a modest degree. Therefore, it is not clear which algorithm attains the higher throughput in general since the results are opposite depending on which vendor's devices are used.

The significant difference in which algorithm has the highest throughput is mainly influenced by the area that the designs require. Although the vendor devices cannot be

directly compared, it can be observed that CubeHash requires more area on Altera and Blue Midnight Wish requires less area on Altera. Therefore, when designing for the same area target, fewer parallel cores of CubeHash are used on Altera. This gives Blue Midnight Wish the advantage. There is also a difference in clock frequency between the vendor devices, but this difference is not as significant as the overall amount of CubeHash cores that fit in the same area as the pipelined Blue Midnight Wish architecture.

Implementing a x1-PAR5 CubeHash design, dividing it to find the area utilization for one core, and then dividing the area of Blue Midnight Wish x1-PPL18 by that one core to find the number *n* for CubeHash x1-PAR*n* was an extremely accurate method. This resulted in very close comparison between the two algorithms. Additionally, the results for CubeHash architectures x1, x1-PAR5, and x1-PAR*n* are very nearly linear in terms of area and throughput. This is true on all vendor devices and both message digest sizes. The conclusion drawn from this is that parallelization of cores truly does attain a predictable performance that can be calculated by using the equation 8.1. In terms of granularity, CubeHash is the clear winner when looking at the Multi-message architectures. When creating parallel architectures, the granularity in terms of area is determined by the size of one core. Since CubeHash is so small, it can be parallelized to a point that nearly exactly fits the resources of the device. The pipelined designs do not have this granularity available because there are no design choices that affect granularity of the size of the design.

$$x1\text{-}PARn(area, throughput) = x1(area, throughput) * n \quad (8.1)$$

In summary, it can be seen that both algorithms do exhibit a wide range of performance space by utilizing the Basic Iterative, Folded, and Multi-message architecture techniques. Blue Midnight Wish has a large amount of control in how much area is saved by the folded design, with a much larger range of granularity than found in CubeHash's folded designs, but CubeHash is so much smaller than Blue Midnight Wish that it would be the preferred algorithm when designing for area. When examining the Basic Iterative designs of both algorithms, Blue Midnight Wish exhibits higher throughput than CubeHash, but the throughput to area ratio of CubeHash is 4 to 8 times higher than Blue Midnight Wish, and the area requirement for CubeHash is significantly lower than Blue Midnight Wish. Finally, both algorithms gain significant throughput by implementing Multi-message architectures. The results show Blue Midnight Wish as having superior throughput on Altera devices, and CubeHash as having superior throughput on Xilinx devices, which is a mixed result. However, the Blue Midnight Wish result is very inflexible since there is no control over the amount of area and maximum throughput attained by the pipelined architecture. In contrast, a designer can vary *n* in *x1-PARn* to utilize an almost arbitrary amount of area, or to reach an almost arbitrary throughput maximum. This is a very important result because it means that CubeHash can be made to completely fill an FPGA device, or can be designed to meet an exact throughput requirement. Blue Midnight Wish would not be able to as easily use a certain amount of area and cannot scale its throughput up or down. In practice, unless

there is an FPGA device that is exactly the same size as the Blue Midnight Wish x1-PPL18 architecture, this means that in a dedicated SHA-3 FPGA device that the chip's area will be underutilized and that a specific throughput requirement will either not be met or may be significantly overachieved. For these reasons, it is found that CubeHash is superior in general due to the flexibility gained from its very small size in conjunction with the properties of multiunit design that it is well suited for.

8.1. Future Work

As stated in the discussion of the CubeHash Multiunit architectures, the most suitable type of algorithms for Multiunit architectures are the lower area design. An extension of this work would be to create Multiunit versions of all five SHA-3 finalists to benchmark their performance and flexibility in this respect, since they are all relatively the same size and are of a medium-low area.

This research applied the techniques of pipelining and parallelization separately. For Blue Midnight Wish the design was too large to place two of them on a device. For CubeHash, pipelining is not practical. If an algorithm could be pipelined significantly and then also remain small enough to be placed in a Multiunit manner, it would hold a significant advantage in terms of maximum throughput flexibility over another algorithm that does not have this property. Therefore, an extension would be to analyze the five SHA-3 finalists to determine if any of them allow this type of architecture.

A discussion of an alternative horizontally folded architecture of CubeHash was given, but was not pursued due to limitations and significant impact to throughput. It

could be possible that this folded architecture could be turned in to a heavily pipelined architecture. This architecture would likely not use the discussed /66(h) architecture which aimed for the lowest area, but instead of working with one 32-bit word at a time, could possibly work with 4 32-bit words at a time for roughly a 10 clock round architecture which would actually be x1-PPL10. This architecture would require more area than x1, but would be expected to exhibit higher throughput than x1 as well. If these were found to be true, a Multiunit pipelined architecture could be created such as x1-PPL10PAR*n*. Future work would be done to determine impacts to protocol, control logic, and other aspects of having pipelined parallel architectures.

Extremely low-area implementations of the algorithms were not done in this study because the architectures are so different to the point that they were outside of the scope of the research. Valuable future work would be a full study and comparison of these architectures to give a clear analysis of the very low area performance space of these algorithms.

REFERENCES

REFERENCES

[1] X. Wang, H. Yu, and Y. Yin, “Efficient Collision Search Attacks on SHA-0,” volume 3621 of LNCS, pages 1-16, 2005.

[2] X. Wang, Y. Yin, and H. Yu, “Finding Collisions in the Full SHA-1,” volume 3621 of LNCS, pages 17-36, 2005.

[3] ——, “Cryptographic Hash Algorithm Competition,” Online, 2011, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[4] ——, “SHA-3 Finalists Announced By NIST,” Online, Dec 2010, http://crypto.junod.info/2010/12/10/sha-3-finalists-announced-by-nist/.

[5] D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, Jorn Amundsen and S. F. Mjolsnes, ”Cryptographic Hash Function BLUE MIDNIGHT WISH,” Submission to NIST (Round 2) of SHA-3 Competition, September 2009

[6] A. Joux, “Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions,” CRYPTO 2004, volume 3152 of LNCS, pages 306–316, 2004.

[7] D. J. Bernstein, “CubeHash Specification (2.b.1).” Submission to NIST (Round 2), 2009.

[8] A. Akin, A. Aysu, O. Ulusel, and E. Savas, “Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa and Blue Midnight Wish for Single- and Multi-Message Hashing,” University Faculty of Engineering and Natural Sciences, Istanbul, Turkey, 2010.

[9] M. El Hadedy, D. Gligoroski, and S. Knapskog, “Single Core Implementation of Blue Midnight Wish Hash Function on VIRTEX 5 Platform,” Norwegian University of Science and Technology, Oct. 2010.

[10] V. Klima, D. Gligoroski, “On Blue Midnight Wish Decomposition,” SantaCrypto 2009, submission.

[11] K. Gaj, E. Homsirikamol, and M. Rogawski, "Comprehensive Comparison of Hardware Performance of Fourteen Round 2 SHA-3 Candidates with 512-bit Outputs using Field Programmable Gate Arrays," SHA-3 Workshop, Santa Barbara, Aug, 2010, submission.

[12] ——, "Athena project website," Online, 2011, http://cryptography.gmu.edu/athena/.

[13] E. Homsirikamol, "Fair and Comprehensive Comparison of Hardware Performance of SHA-3 Round 2 Candidates using FPGAs," Masters Thesis, George Mason University, Virginia, 2010.

[14] ——, "Virtex 7 FPGA Family," Online, 2011, http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/.

[15] E. Homsirikamol, M. Rogawski, K. Gaj, "Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs," CHES, Nara, Japan, Sep. 2011, submission.

CURRICULUM VITAE

Robert Lorentz was born in March of 1982 in Ft. Wayne, Indiana. He received his Bachelor of Science in Computer Science from George Mason University in Fairfax Virginia, graduating *cum laude* in May of 2006. The following year he began his graduate studies in the Electrical and Computer Engineering department of the same school. While performing graduate studies, he has worked full time in industry for 7 years in an engineering role for Rockwell Collins' Aircraft Simulation and Training division located in Sterling, Virginia. His research interests include reconfigurable computing, embedded hardware design, and cryptography.