2011

Versatile FPGA architecture for skein hashing algorithm

David Webster

Follow this and additional works at: <http://scholarworks.rit.edu/theses>

Recommended Citation

Webster, David, "Versatile FPGA architecture for skein hashing algorithm" (2011). Thesis. Rochester Institute of Technology.
Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Versatile FPGA Architecture for Skein Hashing Algorithm

by

David M. Webster

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Prof. Marcin Łukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

Approved By:

Prof. Marcin Łukowiak
Department of Computer Engineering

Prof. Stanisław Radziszowski
Department of Computer Science

Prof. Roy Melton
Department of Computer Engineering

Prof. Alan Kaminsky
Department of Computer Science

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Versatile FPGA Architecture for Skein Hashing Algorithm

I, David M. Webster, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

David M. Webster

Date

Dedication

To family and to my fiancé
Who have always been there to support me in all that I do.

Acknowledgments

Thank you to all of my thesis advisers for their help through this research. Their time and effort is greatly appreciated. Thanks especially to Dr. Lukowiak for bringing this research to my attention and his confidence in me to succeed.

Abstract

Digital communications and data storage are expanding at fast rates, increasing the need for advanced cryptographic standards to validate and provide privacy for that data. One of the basic components commonly used in information security systems is cryptographic hashing. Cryptographic hashing involves the compression of an arbitrary block of data into a fixed-size string of bits known as the hash value. These functions are designed such that it is computationally infeasible to determine a message that results in a given hash value. It should also be infeasible to find two messages with the same hash value and to change a message without its hash value being changed. Some of the most common uses of these algorithms are digital signatures, message authentication codes, file identification, and data integrity.

Due to developments in attacks on the Secure Hash Standard (SHS), which includes SHA-1 and SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512), the National Institute of Standards and Technology (NIST) will be selecting a new hashing algorithm to replace the current standards. In 2008, 64 algorithms were entered into the NIST competition and in December 2010, five finalists were chosen. The final candidates are BLAKE, Keccak, Grøstl, JH, and Skein. In 2012, one of these algorithms will be selected for the Secure Hash Algorithm 3 (SHA-3).

This thesis focuses on the development of a versatile hardware architecture for Skein that provides both sequential and tree hashing functions of Skein. The performance optimizations rely heavily on pipelined and unrolled architectures to allow for simultaneous

hashing of multiple unique messages and reduced area tree hashing implementations. Additional result of this thesis is a comprehensive overview of the newly developed architectures and an analysis of their performance in comparison with other software and hardware implementations.

Contents

Dedication	iii
Acknowledgments	iv
Abstract	v
1 Introduction	1
1.1 Motivation	1
1.2 Skein	2
2 Skein Specification	3
2.1 Threefish	6
2.1.1 MIX and Permute	7
2.1.2 Subkey Generation	8
2.2 Unique Block Iteration	12
2.2.1 Configuration UBI	13
2.2.2 Message UBI	14
2.2.3 Output UBI	15
2.2.4 Tree Hashing	17
3 Supporting Work	20
3.1 Iterative	20
3.2 Unrolled	22
3.3 Pipelined	25

3.4	Duplicate Cores	27
3.4.1	Subkey Generation	29
4	Design Methodology	32
4.1	Versatile Unrolled and Pipelined Architecture	32
4.1.1	Threefish Architecture	33
4.1.2	Deep Pipelined Architecture	37
4.1.3	Subkey Generator	38
4.1.4	Tree Hashing with Pipelined Core	41
4.1.5	Control	45
5	Testing Methodology	49
5.1	VHDL Simulation	49
5.2	Hardware Verification	52
5.3	Synthesis and Implementation Automation	55
6	Performance Results & Analysis	59
6.1	Sequential Hashing	60
6.2	Multiple Message Hashing	61
6.3	Pipelined Tree Hashing Performance	64
6.3.1	Message Processing Overhead	65
6.3.2	Tree Performance	70
7	Conclusions	82
	Bibliography	84

List of Figures

2.1	Threefish block cipher diagram	3
2.2	Tweak parameters diagram [14]	4
2.3	Threefish block in MMO configuration	5
2.4	First four rounds of Threefish-256 [14]	8
2.5	MIX operation	9
2.6	Generation of the first two subkeys for $N_W = 4$	11
2.7	UBIs for simple Skein hashing [14]	12
2.8	Message UBI for M with $3 \cdot N_b$ bytes	15
2.9	Skein with different size outputs	16
2.10	Example of Skein hash tree [14]	17
2.11	Tree structures with different Y_L and Y_F [1]	19
3.1	Iterative Threefish architecture [11]	21
3.2	Eight-round unrolled Threefish architecture [17]	23
3.3	Dual subkey generator [17]	24
3.4	Pipelined architecture [9]	26
3.5	Pipelined architecture with 4 (left) and 8 (right) pipeline registers	28
3.6	MuxSwitch subkey generator [11]	30
3.7	Shift register key generator [17]	31
4.1	Versatile Threefish round architecture	34
4.2	Multiplexed 64-bit adder using single 6-2 LUT	35
4.3	Four-bit adder in single Xilinx SLICEM	36
4.4	Deep pipeline MIX architecture	37
4.5	Subkey generator	40
4.6	Pipelined tree hashing, $Y_L = 1$, $Y_F = 1$, msgBlocks = 16	42
4.7	Pipelined tree hashing, $Y_L = 2$, $Y_F = 2$, msgBlocks = 16	43
4.8	$Y_L = 1$, $Y_F = 4$, msgBlocks = 16	44
4.9	$Y_L = 4$, $Y_F = 4$, msgBlocks = 4096	45
4.10	Controller state machine	46

4.11	Block diagram of complete Skein core	48
5.1	Message storage format	51
5.2	Hardware testing framework	53
5.3	Hardware testing flow	54
5.4	ATHENa results for SHA-3 candidates [8]	56
6.1	Tree structure with different Y_L and Y_F [1]	66
6.2	Message overhead for $Y_F = Y_L, msgBlocks = 8192$	67
6.3	Skein with different size outputs	68
6.4	Message overhead for tree hashing	69
6.5	$Y_L = 4, Y_F = 4, msgBlocks = 4096$	73
6.6	Maximum theoretical speedup of pipelined tree hashing	74
6.7	Maximum theoretical speedup of pipelined tree hashing $Y_F = Y_L$	75
6.8	Maximum theoretical speedup of pipelined tree hashing $Y_F = Y_L$	76
6.9	Maximum theoretical efficiency of pipelined tree hashing	77
6.10	Maximum theoretical efficiency of pipelined tree hashing $Y_F = Y_L$	78
6.11	Theoretical execution time	80
6.12	Theoretical execution time $Y_F = Y_L$	81

List of Tables

2.1	Tweak parameters [14]	4
2.2	Skein type field values	5
2.3	Skein symbol definitions	6
2.4	Threefish variable definitions	7
2.5	MIX rotation values [14]	9
2.6	Permutation values [14]	10
2.7	Skein configuration string [14]	13
2.8	Skein tree hashing symbol definitions	17
3.1	Iterative architecture results	21
3.2	Four round unrolled results	22
3.3	8-round unrolled results	25
3.4	Skein-512 ASIC results (32nm Technology) [9]	27
3.5	Skein-512 pipelined hardware results	27
3.6	Multiple core results [16]	29
3.7	Subkey hardware for Skein-512 [1]	30
4.1	Subkey generator signals for Skein-256	39
4.2	Skein core inputs/output	47
5.1	Skein-256 - Device : Virtex-5 XC5VLX110-3	57
5.2	Skein-256 - Device : Virtex-6 XC6VLX240T-1	57
5.3	Skein-256 - Device : Virtex-6 XC6VLX240T-3	57
6.1	Variables for evaluating Skein hardware performance	59
6.2	Skein 256 Eight-round unrolled results	63
6.3	Calculating the UBI nodes in a Skein tree [1]	67
6.4	Calculation of LAT_L	71
6.5	Calculation of LAT_F	72
6.6	Tree hashing results	79

Chapter 1

Introduction

One of responsibilities of The National Institute of Standards and Technology (NIST) is to determine standards for securing and maintaining the integrity of digital data. In particular the Computer Security Division (CSD) of NIST defines a set of secure hashing algorithms (SHA) in order to accomplish these goals. In 2005 a differential attack on SHA-1 was announced by Prof. Xiaoyun Wang that would discover a hash collisions with complexity 2^{63} rather than the ideal 2^{80} operations [4]. More recent research has determined differential paths with complexity 2^{52} [3]. With this new discovery in addition to collisions on MD4, MD5, SHA-0, and collision attacks on SHA-1, NIST initiated a competition to develop a new hashing algorithm to be named SHA-3 [15]. Since the initial submissions, the field of candidates has been narrowed to five finalists: Keccak, Grøstl, JH, BLAKE, and Skein. Each of these algorithms has been carefully scrutinized by the cryptographic community and NIST for security. The performance of these algorithms with respect to throughput and area on different platforms will also be a factor when selecting the new algorithm for SHA-3.

1.1 Motivation

In attempts to achieve higher performance, FPGAs have been utilized as one of the platforms for implementing the algorithms. Due to Skein's security and being the only finalist

to provide a tree hashing mode, this work has chosen to focus on developing a novel architecture for Skein. There are currently a few different FPGA architectures for Skein, each with its own performance advantages. This flexibility in implementation options makes Skein a suitable choice for hardware development. In addition to being secure Skein's uses basic operations of addition and exclusive OR. These simple operations make FPGAs an ideal target platform. Multiple FPGA architectures have been developed for Skein, and this work intends to contribute to Skein hardware development by developing a more versatile architecture. A detailed evaluation of the performance of this architecture will be provided to aid other researchers in evaluating this design and other future designs.

1.2 Skein

Although the security of Skein is not a focus of this work, its security makes it a strong competitor for SHA-3. NIST's report on the second round candidates stated that the most severe attacks on Skein include a key-recovery attack on 39 rounds of Threefish-256 and 42 rounds of Threefish-512 as well as a rotational distinguisher on 53 and 57 rounds of the Skein-256 and Skein-512 compression functions, respectively [13]. Skein is also one of the most flexible SHA-3 candidates. With its multiple internal state sizes and any size out, it can be used as a direct substitution for any of the current SHA hash functions [14] as well as pseudo random number generator (PRNG), stream cipher, and key derivation function.

Chapter 2

Skein Specification

Skein was developed by a team led by Bruce Schneier and Niels Ferguson and was selected as a finalist for the NIST SHA-3 competition [13]. The base component of Skein is the Threefish tweakable block cipher. The Threefish cipher has three inputs, the plaintext, key, and tweak as shown in Figure 2.1. Using this block cipher allows Skein to operate in a variety of different configurations as well as improve the security of the hashing function.

Figure 2.1: Threefish block cipher diagram

The Threefish function can be represented as $TF(K, P, T)$, with T being the current tweak value, P the current plaintext input, and K the key input. The Skein family of hash functions can use multiple internal state sizes as well as multiple output sizes. The internal state sizes are based off of the internal state size of Threefish. The three standard state sizes for Threefish are 256, 512, and 1024 bits. The output size can be of any length and is independent of the state size. The accepted naming convention is Skein-StateSize-OutputSize. For example the Skein hash function with an internal state size of 256-bits and

output of 256-bits is referred to as Skein-256-256. Since the output size is independent of the internal state size, the variants are designated as Skein-256, Skein-512, and Skein-1024 to represent the internal state size only.

The tweak input to Threefish contains multiple parameters that vary depending on the configuration and message. Figure 2.2 illustrates the tweak value with more detailed descriptions in Table 2.1.

Figure 2.2: Tweak parameters diagram [14]

Table 2.1: Tweak parameters [14]

Name	Bits	Description
Position	0– 95	The number of bytes in the string processed so far (including this block)
reserved	96–111	Reserved for future use, must be zero
TreeLevel	112–118	Level in the hash tree, zero for non-tree computations.
BitPad	119	Set if this block contains the last byte of an input whose length was not an integral number of bytes. 0 otherwise.
Type	120–125	Type of the field (config, message, output, etc.)
First	126	Set for the first block of a UBI compression.
Final	127	Set for the last block of a UBI compression.

The Threefish block cipher is used in a Matayas-Meyer-Oseas (MMO) construction as shown in Figure 2.3 [7]. The output of the MMO (H_i) is equal to exclusive OR of the message block and the output of the Threefish block, which will be described in the Skein Specification section.

Unique Block Iteration is used to chain multiple iterations of the MMO together to hash an entire message and any other UBIs required by the type of hashing being performed.

Figure 2.3: Threefish block in MMO configuration

Each UBI is responsible for processing one of the arguments in Table 2.2, each of which serves a different function. When using multiple arguments, the arguments are processed in increasing order by value.

Table 2.2: Skein type field values

Symbol	Value Range	Description
T_{key}	0	Key (for MAC and KDF)
T_{cfg}	4	Configuration block
T_{pcs}	8	Personalization String
T_{PK}	12	Public key(for digital signature hashing)
T_{kdf}	16	Key identifier (for KDF)
T_{non}	20	Nonce (for stream cipher or randomized hashing)
T_{msg}	48	Message
T_{out}	63	Output

Considering the different sizes and configurations of Skein, several variables are defined to aid in detailing the algorithm. These variables with their range of values and descriptions are shown in Table 2.3. Many of these variables are explained in more detail in following sections.

Table 2.3: Skein symbol definitions

Symbol	Value Range	Description
N_b	32, 64, 128	Internal state size in bytes.
N_o	$1 - (2^{64} - 1)$	Output size in bits
N_M	$1 - (2^{96} - 1)$	Message size in bytes
Y_L	$1 - (2^8 - 1)$	Tree leaf size encoding
Y_F	$1 - (2^8 - 1)$	Tree fan-out encoding
Y_M	$1 - (2^8 - 1)$	Maximum tree height
M	$0 - (2^{99} - 1)$	Message of maximum length $2^{96} - 1$ bytes
G	$0 - (2^8 - 1)$	Chaining variable from previous UBI of N_b bytes.
T_s	See Table 2.1	128-bit initial tweak value

2.1 Threefish

Threefish is the tweakable block cipher used inside each UBI. This block cipher takes three inputs: the key (K) of N_b bytes, the plaintext message input (P) of N_b bytes, and the tweak (T) of 128-bits. Threefish operates internally on 64-bit words, and thus all of the inputs are split into multiple 64-bit words. Each variant of Skein corresponds to the equivalent variant of Threefish (Skein-256 uses Threefish-256, each with $N_b = 32$). The output (C) of Threefish is also a N_b -byte value. The basic representation of the Threefish block cipher was shown in Figure 2.1. From the various inputs, multiple variables can be defined for Threefish. These variables are described in Table 2.4.

Within the Threefish block cipher are three primary operations: MIX, permute, and subkey addition. Each of these operations is described in detail in following sections. A set of $N_w/2$ MIX operations and a single Permute function constitutes a single Threefish round. For Skein-256 and Skein-512, the total number of rounds (N_r) is 72, and 80 for Skein-1024. Before the first round and after every fourth subsequent round, a subkey addition is performed. Figure 2.4 shows the first four rounds and two subkey additions for Skein-256.

Table 2.4: Threefish variable definitions

Variable	Definition
N_w	number of 64-bit words in each message/key block ($8 \cdot N_b/64$)
N_r	number of rounds in the block cipher
i	word index ($0, \dots, N_w - 1$)
j	word index ($0, \dots, N_w/2 - 1$)
d	current round ($0, \dots, N_r - 1$)
$k_0, \dots, k_{N_w-1}$	64-bit words of key input
t_0, t_1	64-bit words of tweak input
$p_0, \dots, p_{N_w-1}$	64-bit words of plaintext input
$v_{d,i}$	i th word of state after d rounds
s	subkey number
p_i	64-bit block of plaintext or message block input
c_i	64-bit block of ciphertext or output block input

2.1.1 MIX and Permute

Each Threefish round contains a number of MIX operations and a single permute operation. The MIX operation is comprised of an unsigned 64-bit addition, a rotation (logical shift), and an exclusive OR. Given two input words (x_0, x_1) , the MIX operation produces outputs (y_0, y_1) according to the following equations.

$$y_0 := (x_0 + x_1) \bmod 2^{64}$$

$$y_1 := (x_1 \lll R_{(d \bmod 8),j}) \oplus y_0$$

A diagram of the MIX function can be seen in Figure 2.5, and Table 2.5 shows the rotation values for all variants of Skein. The rotation value is dependent upon the word index j and round number mod 8 ($d \bmod 8$). Since two words are processed by each MIX block, $N_w/2$ MIX blocks are required for each of the Skein variants. For Skein-256, Skein-515, and Skein-1024, N_w is 4, 8, and 16 respectively, thus requiring 2, 4, and 8 MIX blocks.

After the MIX operation, the 64-bit outputs are reordered before entering the next round. The reordering patterns are shown in Table 2.6. For hardware architectures the

Figure 2.4: First four rounds of Threefish-256 [14]

permutations are achieved simply by routing the outputs of one round to the corresponding inputs of the next round.

2.1.2 Subkey Generation

Before the first MIX and permute and after every fourth subsequent MIX and permute operation, a subkey is added to the current Threefish state. The $N_r/4 + 1$ subkeys are generated using the output of the previous Threefish block (the key), the tweak, and the subkey number. Both the key (K) and tweak (T) are extended by a single word using the formulas shown below. C_{240} is a constant defined as 0x1BD11BDAA9FC1A22. This constant was chosen so that no extended key would be all zeroes. Further explanation of

Figure 2.5: MIX operation

Table 2.5: MIX rotation values [14]

N_w	4		8				16							
j	0	1	0	1	2	3	0	1	2	3	4	5	6	7
$d =$	0	14 16	46 36 19 37	24 13 8 47 8 17 22 37										
	1	52 57	33 27 14 42	38 19 10 55 49 18 23 52										
	2	23 40	17 49 36 39	33 4 51 13 34 41 59 17										
	3	5 37	44 9 54 56	5 20 48 41 47 28 16 25										
	4	25 33	39 30 34 24	41 9 37 31 12 47 44 30										
	5	46 12	13 50 10 17	16 34 56 51 4 53 42 41										
	6	58 22	25 29 39 43	31 44 47 46 19 42 44 25										
	7	32 32	8 35 56 22	9 48 35 52 23 31 37 20										

how this value was determined can be found in the Skein specification [14].

$$k_{N_w} := C_{240} \oplus \bigoplus_{i=0}^{N_w-1} k_i$$

$$t_2 := t_0 \oplus t_1$$

Table 2.6: Permutation values [14]

	$i =$																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
$N_w =$	4	0	3	2	1												
	8	2	1	4	7	6	5	0	3								
	16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

Using the extended key and tweak, each subkey sk_s is generated according to the expressions below.

$$\begin{aligned}
 sk_{s,i} &:= k_{(s+i) \bmod (N_w+1)} && \text{for } i = 0, \dots, N_w - 4 \\
 sk_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + t_{s \bmod 3} && \text{for } i = N_w - 3 \\
 sk_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + t_{(s+1) \bmod 3} && \text{for } i = N_w - 2 \\
 sk_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + s && \text{for } i = N_w - 1
 \end{aligned}$$

This subkey generation can be more easily understood through the illustration in Figure 2.6. This figure shows that after each subkey is generated, the extended key and tweak are shifted one word to the left, and the additions are performed as before. There are two standard approaches to implementing the subkey generator in hardware, both of which are discussed in the supporting work section.

Figure 2.6: Generation of the first two subkeys for $N_W = 4$

2.2 Unique Block Iteration

Unique Block Iteration is the chaining of multiple Threefish blocks together in order to compress an arbitrary length input to a fixed size output. Skein uses UBIs to execute the processing for each of the arguments of Skein shown in Table 2.2. Each UBI has three inputs and a single output. The inputs to a UBI are the message (M) of arbitrary length with a maximum length of $(2^{96} - 1)$ bytes, an initial chaining variable (G_0) of N_b bytes, and an initial tweak value of 128-bits (T_s). The output of any UBI is also of size N_b bytes. The UBI for each of these arguments contains a different number of Threefish blocks depending on the size of the message input. A UBI can be represented by the expression shown in Equation 2.1. This work focuses on simple Skein hashing, which utilizes only the Configuration, Message, and Output UBIs as shown in Figure 2.7. The remaining UBI types are detailed in the Skein specification [14].

$$G_i = UBI(G_{i-1}, M, T_s) \quad (2.1)$$

Within a UBI, multiple Threefish blocks may be chained together. When multiple blocks are chained together, the fields in the tweak are varied to customize the particular block of M being processed.

Figure 2.7: UBIs for simple Skein hashing [14]

2.2.1 Configuration UBI

The configuration UBI is the first UBI to be processed for a simple Skein hash and is required for all methods of Skein hashing. The message input to this UBI is a configuration string of N_b bytes. The configuration string contains multiple parameters, which are shown in Table 2.7. Several parameters are variable and define the form of hashing being performed. The remaining parameters are currently fixed values, which include the schema identifier, which is simply the ASCII string “SHA3”, the version number, which is currently 1, and multiple reserved reserved bytes set to 0. The variable parameters include N_o indicating the output length being used, and the tree configuration parameters (Y_L , Y_F , Y_M), which determine the leaf size, fanout, and maximum height of the constructed tree. The tree mode parameters are explained in more detail in the Tree Hashing section.

Table 2.7: Skein configuration string [14]

Offset	Size in Bytes	Name	Description
0	4	Schema identifier	The ASCII string “SHA3” = (0x53, 0x48, 0x41, 0x33)
4	2	Version number	Currently set to 1
6	2		Reserved, set to 0
8	8	Output length	N_o
16	1	Tree leaf size enc.	Y_L
17	1	Tree fan-out enc.	Y_F
18	1	Max. tree height	Y_M
19	13		Reserved, set to 0

Using the configuration string, one can define the UBI using the following inputs and the output defined in Equation 2.2.

$G := K'$, Output of Key UBI or string of N_b zero bytes

$M := C$, Configuration string as defined in Table 2.7

$T_S := T_{cfg}$, Initial tweak value.

$$G_0 = UBI(K', C, T_{cfg} 2^{120}) \quad (2.2)$$

2.2.2 Message UBI

After the configuration UBI, the message to be hashed is processed. In the message UBI multiple Threefish blocks are used in order to process the entire N_M bytes of the message. Since an entire block of N_b bytes is required for the plaintext input of each Threefish block, a message that is not an integer multiple of 8 bits must be padded by setting the most significant unused bit to “1” and the remaining unused bits to “0” until a multiple of N_b bytes is achieved. The padded message is split into $\lceil N_M/N_b \rceil$ message blocks ($M_0, \dots, M_{k-1}$), and therefore equally as many Threefish blocks are chained together to complete the message UBI. For sequential Skein hashing the message UBI is processed using the following inputs and output defined in Equation 2.3.

$G := G_0$, Output of Configuration UBI

$M := M$, Message to be hashed of length N_M bytes

$T_S := T_{msg}$, Initial tweak value.

$$G = UBI(G_0, M, T_{msg} 2^{120}) \quad (2.3)$$

Figure 2.8 shows an example of a message UBI for a message with exactly $3 \cdot N_b$ bytes. The chaining variable G_0 is the output from the configuration UBI. The message input is the first block (M_0) of the message, and the initial tweak T_0 has the first block field set to “1” and the position is set to N_b . The key input to the next two Threefish blocks is the output of the previous Threefish block. For T_1 , the first block field is reset to “0” and the position is set to $2 \cdot N_b$. The last tweak T_2 sets the last block field to “1” and the position to $3 \cdot N_b$. H_2 would become the chaining variable input for the output UBI. For a message that is not an integer multiple of N_b bytes before padding, the position field of the last tweak is set to the to the number of bytes prior to padding.

Figure 2.8: Message UBI for M with $3 \cdot N_b$ bytes

2.2.3 Output UBI

The final stage of processing is the Output UBI. This stage takes the chaining variable from the Message UBI and generates the N_o bits for the final hash value. If N_o is equal to or less than that of $8 \cdot N_b$, only one output UBI is required to generate the final output of Skein. If N_o is greater than $8 \cdot N_b$, $\lceil N_o / (8 \cdot N_b) \rceil$ UBIs are needed to compute the final output of Skein. The inputs to the Output UBI are defined as follows :

$G := G$, Output of Message UBI

$M := 0, 1, 2, \dots$, Byte value corresponding to which Output UBI is being processed

$T_S := T_{out}$, Initial tweak value.

The final hash output is the first N_o bits of H as defined in Equation 2.4 are used.

$$H = UBI(G, 0, T_{out}2^{120}) || UBI(G, 1, T_{out}2^{120}) || UBI(G, 2, T_{out}2^{120}) || \dots \quad (2.4)$$

Examples of Skein with different output sizes are illustrated in Figures 2.9(a) and 2.9(b). Figure 2.9(a) shows how the UBIs of Skein are chained together in the case of $N_o \leq 8 \cdot N_b$, which requires only one Output UBI. In the second example $2 \cdot N_b \cdot 8 \leq N_o \leq 3 \cdot N_b \cdot 8$,

and thus three Output UBIs are required as illustrated in Figure 2.9(b).

(a) $N_o \leq 8 \cdot N_b$

(b) $2 \cdot N_b \cdot 8 \leq N_o \leq 3 \cdot N_b \cdot 8$

Figure 2.9: Skein with different size outputs [14]

2.2.4 Tree Hashing

In sequential Skein, the message is processed in a single UBI. In order to allow for the ability to speed up the message processing, the designers of Skein included a tree hashing option. In tree hashing, the message UBI can be split up into multiple UBIs to construct a tree. A diagram depicting a generic hash tree is shown in Figure 2.10.

Table 2.8: Skein tree hashing symbol definitions

Symbol	Description
Y_L	The leaf size encoding. Number of blocks processed by each leaf node is 2^{Y_L} , $Y_L \geq 1$
Y_F	The fan-out encoding. The fan-out of a tree is 2^{Y_F} with $Y_F \geq 1$
Y_M	The maximum tree height; $Y_M \geq 2$, 255 if not specified
N_l	Leaf size, $N_b 2^{Y_L}$
N_n	Node size, $N_b 2^{Y_F}$

The structure of the tree is defined by Y_L , Y_F , and Y_M . These parameters are defined along with other important variables for tree hashing as shown in Table 2.8.

Figure 2.10: Example of Skein hash tree [14]

For every UBI in the tree, the chaining variable (G) is the output of the previous

stage of processing, usually the T_{cfg} stage. Equations 2.5, 2.6, and 2.7, describe the tree processing of a message (M). The message is initially split into $k = \lceil N_M / (N_b \cdot 2^{Y_L}) \rceil$ message blocks ($M_{0,0}, \dots, M_{0,k-1}$) of size $N_b \cdot 8 \cdot 2^{Y_L}$ bytes for $M_{0,0}$ through $M_{0,k-2}$ and less than or equal to $N_b \cdot 8 \cdot 2^{Y_L}$ bytes for $M_{0,k-1}$. The outputs of the leaf level UBIs are concatenated to produce a new message M_1 (Eq. 2.5). This new message is then split into j blocks of size $N_b \cdot 2^{Y_F}$. Each subsequent level is processed (Eq. 2.6) until the root node is reached. The root node produces the chaining variable to the output UBI (G_0) using Equation 2.7. The tweak for each of the following equations takes into consideration the tree level, and the position field now indicates how many bytes have been processed by the UBIs at the current level.

$$M_1 = \parallel_{i=0}^{k-1} UBI(G, M_{0,i}, T) \quad (2.5)$$

$$M_{l+1} = \parallel_{i=0}^{j-1} UBI(G, M_{l,i}, T) \quad (2.6)$$

$$G_o = UBI(G, M_{Y_M-1}, T) \quad (2.7)$$

Varying the parameters Y_L , Y_F , and Y_M produces different tree structures for a given message. Figure 2.11 illustrates different tree structures for a message of $8 \cdot N_b$ bytes. In each of the examples Y_F and Y_L are varied. For $Y_L = 1$, the leaf level will contain 4 UBIs as each UBI will process 2^{Y_L} message blocks, and for $Y_L = 2$ each leaf level UBI can process 4 message blocks. Each subsequent level processes 2^{Y_F} outputs from the previous level. If a maximum tree height (Y_M) is defined, once the tree reaches this height, the root node will process all UBI outputs of the previous level. If Y_M is undefined, the default maximum tree height is 255.

Figure 2.11: Tree structures with different Y_L and Y_F [1]

Chapter 3

Supporting Work

When designing FPGA architectures for Skein, there are many different factors to consider such as maximum clock frequency, latency, area, and the resulting throughput. Early designs focused on methods of improving the sequential Skein hashing but recent research has developed more novel architectures. In [1], Aric Schorr developed a multicore architecture in order to utilize Skein’s tree hashing mode, while in [9] Walker, Sheikh, Mathew, and Krishnamurthy developed an application specific integrated circuit (ASIC) capable of hashing multiple unique messages simultaneously. This work furthers the above mentioned work to develop flexible and high-performance architectures utilizing unrolled and pipelined architectures to perform sequential, multiple message, and tree hashing. This is accomplished without any Configurable Logic Block (CLB) overhead in the primary Threefish compression function core.

3.1 Iterative

The first FPGA architecture developed for Skein by Men Long was described in [11]. The main focus of his work was the iterative architecture, which is the most straightforward design as only one round is implemented in hardware. This round contains the 64-bit adders and exclusive ORs for the MIX function. The dynamic rotations of Threefish require multiplexers to switch between the different possible rotations. A set of $8 \cdot N_b$ registers is then required to hold the current state of Threefish. Due to the registering of the current

state of Threefish after each round, the latency of the iterative architecture is 72 clock cycles for Skein-256 and Skein-512, and 80 cycles for Skein-1024. At the input to the Skein round hardware is a multiplexer that selects between the message input block or the current Threefish state. The exclusive OR at the output of the round logic produces the chaining value or hash output as required by the MMO construction. The last component of the iterative hardware is the subkey generator. The core iterative architecture used to perform Skein hashing is shown in Figure 3.1. Although the iterative architecture is not used in this work, important design considerations regarding subkey generation can be gained from the work of Long [11] and Schorr [1] in order to achieve the highest performance for the architecture presented in this work.

Figure 3.1: Iterative Threefish architecture [11]

Table 3.1: Iterative architecture results

Design	Device	Bits	Slices	Clk (MHz)	Latency (Cycles)	TP (Mbps)
[11]	XC5VLX50-3	256	1001	114.9425	72	408.68
[11]	XC5VLX50-3	512	1877	114.9425	72	817.37
[11]	XC5VLX50-3	1024	3656	84.8176	80	1085.67

3.2 Unrolled

Although the iterative approach is the most straightforward, the performance results are not satisfactory. One of the bottlenecks is the multiplexers required to implement the dynamic rotations in the MIX operation. In order to eliminate the delay incurred through the multiplexers, designers can unroll the hardware to reduce the size or completely eliminate these multiplexers [17]. To determine how many times to unrolled the hardware, specific elements of the Skein algorithm are examined. The first option is to unroll the hardware 4 times, since subkeys are added to the Threefish state every 4th round. This architecture requires 4 full MIX and permute operations to be implemented in hardware, and the subkey generator remains the same. Although multiplexers are still required to handle the dynamic shifts, their size is greatly reduced from 8-1 multiplexers to 2-1 multiplexers. Unrolling 4 rounds increases the critical path and thus lowers the maximum achievable clock frequency, but in turn decreases the latency to 18 clock cycles [11]. Long's results along with the results of other 4-round unrolled architectures are shown in Table 3.2. Long's throughput is improved through the unrolling, but clearly the his subkey generation architecture still hinders this design which is discussed in a later section.

Table 3.2: Four round unrolled results

Design	Device	Bits	Slices	Clk (MHz)	Latency	TP (Mbps)
[11]	XC5VLX50-3	512	2662	40.8	18	1161
[5]	Virtex-5*	512	1716	119.1	19	3209
[10]	XC5VLX30-3	256	854	115.0	21	1482
[2]	Virtex-5*	512	1786	83.7	22	1945
* Unspecified Virtex-5 FPGA						

Further improving upon the 4-round unrolled approach, Tillich's 8-round unrolled design has been the inspiration for most of the recent unrolled designs [17]. By unrolling 8 rounds, all dynamic shifts of the MIX operation are eliminated and these shifts are accomplished simply by the routing of the wires. Similar to the 4-round unrolled approach,

additional hardware is required to implement the extra rounds; however the subkey generator must also be modified. Since a subkey must be added every 4th round, two subkeys must be generated in each clock cycle. Additional hardware is required to generate both subkeys each clock cycle. Tillich's 8-round unrolled architecture and dual subkey generator are shown in Figures 3.2 and 3.3 respectively.

Figure 3.2: Eight-round unrolled Threefish architecture [17]

Figure 3.3: Dual subkey generator [17]

Once again, the critical path is increased, and the latency is further decreased to 10 clock cycles. The performance of 8 round unrolled architectures has proven to be the most efficient solution for sequential hashing on FPGAs. Results for several different 8 round unrolled architectures are shown in Table 3.3.

Table 3.3: 8-round unrolled results

Design	Device	Bits	Slices	Clk (MHz)	Latency (Cycles)	TP (Mbps)
[17]	XC5VLX110-3	256	937	68.4	10	1751.0
[17]	XC5VLX110-3	512	1632	69.0	10	3534.8
[17]	XC5VLX110-3	1024	2994	68.9	11	6414.0

3.3 Pipelined

A pipeline Skein architecture was first introduced by Walker et al. in [9] shows benefits of pipelining unrolled architectures through an Application Specific Integrated Circuit (ASIC) design. When pipelining an unrolled Skein architecture, registers are inserted between each round. In turn, the critical path is decreased thus increasing the achievable clock frequency. However, due to data dependencies, hashing a single message with a pipelined design will see a decrease in performance as a pipelined design increases latency. A pipelined design allows for multiple unique messages to be hashed simultaneously. This type of architecture would be especially beneficial in a system where multiple data stream must be processed at the same time such a network router [9]. This establishes that a pipelined hardware architecture can be efficiently utilized in certain applications. Figure 3.4 shows the high level diagram of the pipelined architecture.

Although the most obvious application of a pipelined architecture is to hash multiple messages simultaneously, it has been proposed in [11] and in [1], that tree hashing could also benefit from a pipelined architecture. Since tree hashing splits a single message UBI into multiple UBIs, it would be possible to execute multiple UBIs simultaneously using a pipelined core. This approach could reduce the hardware required in comparison with the

duplicate core model [1].

Figure 3.4: Pipelined architecture [9]

The results for the ASIC design are shown in Table 3.4, and although performance of this circuit can not directly be compared to FPGA performance, the area overhead percentage and speedup for each of the pipelined architectures in [9] can give some insight as to what to expect in a pipelined architecture on an FPGA. The design with only one pipeline register is equivalent to an 8 round unrolled architecture. The 2, 4, and 8 pipeline registered architectures insert a register every fourth, every other, and every round respectively. Simplified diagrams of the 4 and 8 pipeline registered architectures are show in Figure 3.5.

In [6] several different pipelined Skein FPGA implementations were developed for Skein-512. These implementations include a 4-round unrolled architecture with two pipeline registers, a 4-round unrolled architecture with 5 pipeline register, an 8-round unrolled architecture with 10 pipeline registers similar to the architecture developed in this work. The

results for these designs is shown in Table 3.5

Table 3.4: Skein-512 ASIC results (32nm Technology) [9]

Pipeline Registers	Area (Gates)	Clk (MHz)	Latency (Cycles)	Throughput (Gbps)	Speedup	Area Overhead (%)
1	57,931	631.31	10	32.32	1.00	0.00
2	60,395	1126.13	20	57.66	1.78	4.25
4	62,954	1736.11	40	88.89	2.75	8.67
8	70,071	2380.95	80	121.90	3.77	20.96

Table 3.5: Skein-512 pipelined hardware results

Design	Architecture	Slices	Throughput (Mbps)
4/2 [6]	Virtex5	2314	5013
4/5 [6]	Virtex5	3942	6141
8/10 [6]	Virtex5	8831	10973
4/2 [6]	Virtex6	1818	5649
4/5 [6]	Virtex6	3209	7664
8/10[6]	Virtex6	7323	11982

3.4 Duplicate Cores

An additional architecture for Skein is aimed at efficiently processing messages using Skein tree hashing. This architecture, developed by Aric Shorr [1], uses multiple unrolled cores to process multiple UBIs of the message stage of Skein simultaneously. Each core is connected to a central controller which assigns each core a particular UBI to process. This architecture was designed for scalability such that only non-bus signals are added when the number of cores is increased. Performance results of this architecture are shown in Table 3.6.

The handling of the message in memory was also carefully examined, and a memory storage scheme was developed to coordinate the reading and writing of UBI outputs

Figure 3.5: Pipelined architecture with 4 (left) and 8 (right) pipeline registers

Table 3.6: Multiple core results [16]

Device	Input block size (bits)	Slices	Clk (MHz)	Latency (Cycles)	Throughput (Mbps)	Number of cores
XC5VLX110-3	256	3024	59.3	10	3037	2
XC5VLX155-3	256	5657	54.9	10	5625	4
XC5VLX155-3	512	5454	50.1	10	5129	2

to/from memory. It was also suggested in [1], that a pipelined architecture could be used for implement tree hashing. This concept is thoroughly explored in this work.

3.4.1 Subkey Generation

There are two different ways to implement the key scheduler. In [11], Long uses a MuxSwitch approach to perform the “shifting” of key and tweak words by selecting the appropriate key and tweak words to feed into the subkey adders. The second method uses shift-registers to shift the key and tweak words while the inputs to the subkey adders remain connected to the same registers. This method was first introduced by Tillich [17]. It is important to note that the registered approach also adds an additional cycle to the latency, increasing it to 73 cycles for Skein-256 and Skein-512, or 81 cycles for Skein-1024 for the iterative approach, or 10 and 11 cycles respectively for the 8-round unrolled design. The performance ramifications of the MuxSwitch approach are that flip-flops are saved, but the look up tables (LUTs) required to construct the multiplexers introduce significant delays.

In [1], the impact on the throughput and sizing of the iterative architecture due to these two different subkey generator implementations was analyzed, and the results are shown in Table 3.7.

Figure 3.6: MuxSwitch subkey generator [11]

Table 3.7: Subkey hardware for Skein-512 [1]

Design	Device	Bits	Slices	Clk (MHz)	% Increase
MuxSwitch	XC5VLX110-3	512	818	NA	
MuxSwitch (Unrolled)	XC5VLX110-3	512	1263	NA	54.40
Registered	XC5VLX110-3	512	448	1574.8	
Registered (Unrolled)	XC5VLX110-3	512	480	1564.9	7.14

Figure 3.7: Shift register key generator [17]

Chapter 4

Design Methodology

While designing the architecture for this work, many factors were considered such as latency, throughput, and area. The main goal of this work was to create a versatile FPGA architecture that can hash messages in multiple modes while maintaining high throughput and small area. Efficient utilization of the FPGA hardware was key when designing the architecture. Previous design concepts and performance analysis aid in the development of the proposed architecture. The architecture is designed to be fully autonomous and assumes an ideal memory interface is provided, which will be explained in later sections. For discussion on non-ideal memory interfaces for Skein hardware see [1]. While the concepts presented in the following sections can be applied to any of the variants of Skein, this work focuses only on Skein-256-256.

4.1 Versatile Unrolled and Pipelined Architecture

In order to create an architecture capable of hashing messages in multiple modes, the core component of Skein, Threefish, must be modified to accommodate these modes. The first mode that the Threefish core must be capable of operating in is the unrolled sequential mode as in [1] and [17]. This is the basis for the versatile Skein architecture. The second mode, a multiple message mode, requires a pipelined Threefish core as seen in [9] that is capable of hashing multiple messages simultaneously. The last mode, the pipeline tree hashing mode utilizes the same hardware as the multiple message mode with a different

control scheme. Merging these unrolled and pipelined Threefish architectures will create the versatile Threefish compression function.

4.1.1 Threefish Architecture

The hardware developed in this work takes the standard unrolled Threefish hardware architecture and inserts pipeline registers between each round and after each subkey addition. This modification creates a pipelined design with 10 pipeline stages. This pipeline differs slightly from that in [9] in that there are pipeline registers after each subkey addition; thus no more than one 64-bit addition takes place in a single clock cycle. The insertion of these additional registers is done in order to increase the speedup achieved through pipelining. From [1], the iterative model with a critical path containing two 64-bit adders achieves a maximum clock frequency of 176 MHz, whereas the unrolled model achieves a maximum clock frequency of 69 MHz. Without adding pipeline registers after each subkey addition, the maximum speedup achieved through pipelining only would be 2.55. By reducing the critical path to a single 64-bit adder, a speed up in clock frequency of 4–5 should be expected. By adding these pipeline stages the latency for the full 72 rounds of Threefish-256 is increased to 91 cycles for the first message being hashed. Additional cycles of latency are incurred for the additional pipeline stages and overhead cycles.

By pipelining the Threefish core, it is capable of hashing multiple messages simultaneously but reduces the performance of hashing only a single message because of the increased latency [9]. To maintain the functionality of the standard unrolled Threefish architecture, the input to each round/subkey adder is multiplexed between the combinational and registered outputs of the previous round/subkey. These multiplexers allow the hardware to switch between registering the internal state after every round/subkey addition or only registering after the first subkey adder. Figure 4.1 shows the versatile round architecture for a single round of Threefish-256.

To utilize the FPGA’s hardware efficiently, the multiplexers for the inputs to MIX function are integrated into the LUTs that perform the MIX operation rather than implemented

Figure 4.1: Versatile Threefish round architecture

in their own LUTs so that no overhead is incurred. Figure 4.2 illustrates how the 6-input 2-output LUTs in the Xilinx Virtex-5 and Virtex-6 FPGAs are used to integrate these multiplexers.

Figure 4.2: Multiplexed 64-bit adder using single 6-2 LUT

SLICEM

Figure 4.3: Four-bit adder in single Xilinx SLICEM

Each slice in a Virtex-5 or Virtex-6 FPGA contains 4 LUTs and a dedicated carry-chain for fast carry-bit propagation, and thus a single slice can implement 4 bits of addition as shown in Figure 4.3. The dedicated carry-chain of each slice is connected to the adjacent slices above and below it in order to create the larger adders.

4.1.2 Deep Pipelined Architecture

In an attempt to reduce the critical path of the architecture further, a “deep-pipelined” architecture is proposed. This deep-pipelined architecture splits the 64-bit additions into two 32-bit addition separated by registers. This further pipelines the Threefish compression function creating a total of 20 pipeline registers. Figure 4.4 illustrates how the adders in the Threefish round can be split into two 32-bit additions. Although the latency of this new deep-pipelined architecture is doubled, the increase in maximum clock frequency due to the decreased adder delay will provide increased performance when hashing many messages simultaneously and in tree mode.

Figure 4.4: Deep pipeline MIX architecture

4.1.3 Subkey Generator

The second main component of the Threefish cipher is the subkey generator. As detailed in the specification section, the subkey generator produces a subkey that is added to the current state of Skein before the first round and after every 4th round. In order to accommodate both the sequential and pipelined Threefish, a specialized subkey generator was developed. In particular two individual subkey generators are necessary to produce the subkeys for the unrolled version of Threefish. The pipelined Threefish compression function in this work contains 10 pipeline registers and thus can hash up to 10 messages simultaneously, and up to 20 messages for the deep pipelined Threefish core. The subkey generator must then store the extended key and extended tweak for all 10 messages or 20 messages for the deep-pipelined architecture. As stated earlier, the highest performing approach to subkey generation is to use registers to store the extended key and extended tweak. For 10 messages, this would require $10 \cdot (320 + 192) = 5120$ flip flops, which would require a minimum of 640 slices in a Virtex-6 or 1280 slices in a Virtex-5 as each slice contains 8 or 4 flip-flops respectively for each of the subkey generators [20][19]. Twice the number of registers would be required for the deep pipelined architecture. Implementing the subkey generator in this way would be a very area inefficient method to implement the subkey generator. To save resources, FIFOs are used in place of registers to store the extended key and extended tweak words. To implement these FIFOs, Xilinx Distributed RAM FIFOs were used. Distributed RAM utilizes the LUTs as memory, thus reducing resource usage. There are two main reasons for using distributed RAM. First, the synthesis tools can better optimize the design and minimize routing delays as the location of the data is not locked into a particular area of the FPGA. Second, since distributed RAM uses the slice logic, direct area comparisons can be made to previous work.

Also observing the subkey architecture from [17], unrolling Threefish 8 rounds requires two subkeys to be generated at the same time for sequential hashing of a single message. To support the generation of both subkeys, the generator must be capable of shifting the

extended key and tweak words by 1 word for pipelined Threefish and by 2 words for sequential mode. Descriptions of the inputs and outputs to the subkey generator are shown in Table 4.1.

Table 4.1: Subkey generator signals for Skein-256

Signal	Description
k(255:0)	Initial key (chaining value)
t(127:0)	Initial tweak value
init_subkey_en	Asserted at beginning of new message block to load in key
next_subkey	Asserted to read key and tweak words from FIFOs and generate next subkey. Also signifies write shifted words back into FIFO if not last subkey.
last_subkey	Asserted when generating last subkey, key and tweak words are not written back into FIFOs.
shift_two_words	Tweak and key words are shifted twice when “1” else shifted once.

It is also important to note that in order to improve performance, an additional register was placed at the input to the FIFO. This register reduces the delay from the output of one FIFO to the input of another. The extra cycle affects only the generation of the first subkey adding one cycle to the latency of the pipeline Threefish and no extra cycles to the latency for sequential Threefish.

Figure 4.5: Subkey generator

4.1.4 Tree Hashing with Pipelined Core

The pipelined unrolled core is ideal for hashing multiple messages simultaneously to increase throughput but there is another advantage to this architecture. Since the pipelined Threefish compression function can process multiple UBIs concurrently, the tree hashing mode can be implemented using this same architecture without duplicating the cores. In [1], multiple sequential cores were utilized to process multiple UBIs for the message at once. The duplicate cores increase the overall area of the architecture significantly. In order to illustrate how the pipelined architecture can implement tree hashing a few examples are shown in Figures 4.6, 4.7, 4.8 and 4.8. In the first example shown in 4.6, a message with 16 blocks and tree parameters of $Y_L = 1$ and $Y_F = 1$ is hashed in pipeline mode. In this example all UBIs in the leaf level can be hashed in the latency of two Threefish blocks or $LAT_{PIPE} * 2^{Y_L} = 2 * LAT_{PIPE}$ since the pipelined core is capable of processing up to 10 independent UBIs simultaneously. The timing diagrams illustrate which UBIs are processed during a particular Threefish iteration. $UBIX_i$ indicates that the Xth UBI is processing its i^{th} message block. The remaining levels are processed in the same way. For the same message, the tree parameters are changed to $Y_L = 2$ and $Y_F = 2$ in Figure 4.7. Although the parameters are changed, the total time is not increased since each level now takes $LAT_{PIPE} * 2^{Y_L} = 4 * LAT_{PIPE}$ cycles but the number of levels was cut in half. However, in Figure 4.8, $Y_L = 1$ and $Y_F = 4$, and the total execution time is increased because of the larger tree fanout. For small messages it is apparent that pipelined tree hashing offers no benefit. However in Figure 4.9, the message size is much large at 4096 blocks. This example shows that all of the pipeline stages are filled much more often resulting in a significant decrease in processing cycles. Sequentially the message would take $4096 \cdot LAT_{SEQ}$, but with the pipelined tree hashing it only takes $464 \cdot LAT_{PIPE}$ which is significantly less time. A detailed analysis of the effect of these parameters on pipeline tree hashing is presented in the Performance Results and Analysis section. Additionally, with a pipelined architecture, an entire message block must be delivered to the core every clock cycle. In this work an ideal memory interface that is capable of delivering a 256-bit

message block each clock cycle is used in order to achieve the best performance.

Figure 4.6: Pipelined tree hashing, $Y_L = 1$, $Y_F = 1$, msgBlocks = 16

Figure 4.7: Pipelined tree hashing, $Y_L = 2$, $Y_F = 2$, msgBlocks = 16

Figure 4.8: $Y_L = 1$, $Y_F = 4$, msgBlocks = 16

Figure 4.9: $Y_L = 4$, $Y_F = 4$, msgBlocks = 4096

4.1.5 Control

For this architecture to be fully autonomous, a control unit must be developed for the Threefish core. The state machine shown in Figure 4.10 is at the heart of the control unit. When the Skein core receives the coreStart signal indicating that the messages are ready in memory, the state machine advances from the IDLE state and enters the INIT state. In this state, the number of blocks, original byte count, and flag indicating the message was padded, for each of the message(s) is read from memory and stored. The state machine then advances to one of the three hashing states, SEQ, MULTIPLE_MSG, or TREE based on the MODE input while asserting the ubiStart signal to begin the first UBI. Each of the message(s) is hashed in this state. If all messages are complete or currently in the last iteration of Threefish (Output UBI for sequential and multiple message hashing, or root node message UBI for tree mode), the state machine advances to the LAST_TF state. Once the final Threefish iteration is complete, the state machine moves to the DONE state, asserts coreDone, and returns to IDLE.

Additionally in the control unit there are two FIFOs. These FIFOs are used to store the input message blocks so that they are available at the end of a Threefish operation to XOR with the output. The resulting hash must also be stored in a FIFO and read as the chaining

Figure 4.10: Controller state machine

value at the beginning of the next Threefish iteration. A complete block diagram of the Skein core is shown in Figure 4.11. Table 4.1.5 lists the inputs and outputs to the top level core.

Table 4.2: Skein core inputs/output

Inputs:		
clk, reset	wire	
coreStart	wire	Indicates message(s) is/are ready and to start hashing
mode	vector(2)	Value indicating what mode to operate in (“00” for sequential mode, “01” for multiple message mode, and “11” for tree hashing mode)
numMessages	vector(5)	Indicates the number of messages to hash from memory when operating in multiple message mode
msgBlock	vector($N_b * 8$)	Message block from memory
Outputs:		
msgBlockAddr	vector(12)	Address to message memory
digestValid	vector(<i>pipelineRegs</i>)	One-hot vector indicating which messages’ result is ready
digestValue	vector($N_b * 8$)	Message digest
coreDone	wire	Indicates all message(s) have been hashed.

Figure 4.11: Block diagram of complete Skein core

Chapter 5

Testing Methodology

Due to the complexity of the architecture proposed in this work, multiple testing strategies were required to verify the architecture. A comprehensive set of behavior and post place and route simulations were used to model the hardware architecture accurately before hardware verification. The testbenches were designed to resemble the hardware architecture, and standardized test vectors were used in both simulations and hardware testing.

5.1 VHDL Simulation

Before testing the design on an FPGA, it had to be fully verified in simulation. The simulation framework from [1] was modified to accommodate the different modes of hashing and the new memory storage scheme from this work. Rather than use a simple simulation variable to store the messages to be hashed, an embedded memory was included in the testbench in order to act much like the hardware testing framework would. The embedded memories are dedicated memories on an FPGA designed to store larger amounts of data, reducing LUT utilization. A memory storage scheme similar to that proposed in [5] was utilized for both the simulation and hardware verification of the architectures developed in this work. Since multiple message hashing is a mode of this hardware architecture, the embedded memory is divided to provide space for multiple messages in a single memory. For the simulation, the testbench reads the initial chaining value, message bit length, number of blocks, the message itself, and correct message digests from the stimulus file generated

from the KAT test vector file. The stimulus file is generated using the JAVA test file generator from [1]. Using the bit length and number of blocks, the testbench calculates if the last block was padded or not. If the last block is padded, a “1” is written into the most significant bit of the first block of a page of memory. Additionally, the number of blocks (*msgBlocks*) in the message and total number of message bytes (N_M) before padding are also written to the first block of a page of memory. The message is then written into the remaining space of memory for that page. Figure 5.1 illustrates the organization of data in the embedded memory. Since the largest test message used is 134 blocks, the memory is divided into 256 blocks per page. For the 8-round unrolled and pipelined design, up to 10 messages need to be stored, and thus a 4K deep embedded is needed. For the 8-round unrolled and deep-pipelined architecture, 20 messages must be stored into the memory; an 8K deep embedded is used to accommodate this.

Behavioral simulations tested each mode individually for all 2561 test vectors generated from the ShortMsgKAT and LongMsgKAT files provided by NIST and the developers of Skein. Also, simulations were run with dynamic switching between modes as follows: 1 message hashed sequentially, 10 hashed simultaneously, and 1 hashed in tree mode repeating until all test vectors were completed.

In addition to the behavioral simulations, post place and route simulations were executed. Post place and route simulations are used to simulate the actual propagation delays in an FPGA. In order for the sequential mode to operate properly, a multicycle path constraint is necessary so that synthesis tools are aware that the unpipelined data path registers are not enabled every clock cycle. All the multiplexers in the Threefish core are connected to the same control signal, and all select either the registered or unregistered input. However, the synthesis tools make no assumptions about which input to each multiplexer will be selected during normal operation. All combinational paths that including those that will not ever be selected during operation are analyzed by the synthesizer. To ignore the paths not used during normal operation, false path constraints must also be added. Due to the number of false paths and the widths of the false data paths, the synthesis tools are not

Figure 5.1: Message storage format

capable of processing these constraints in a reasonable amount of time or with a reasonable amount of memory. The post place and route simulations are used to verify the timing of the architecture before selecting a clock frequency for the hardware verification. A subset of test vectors were selected for the post place and route simulations as the execution time is much longer.

5.2 Hardware Verification

To verify the design in hardware, a testing framework was developed for the Xilinx ML605 development board. This board is based around the Virtex-6 VC6VLT240T-1 FPGA. Rather than build the hardware testbed from the ground up using VHDL, a soft-processor was used to provide the Skein core with messages and control signals, as well as to verify the results from the core. (A soft-processor is a processor that uses the logic on an FPGA to build a traditional microprocessor.) For this work the Xilinx Microblaze core was used. Microblaze is a 32-bit reduced instruction set computer (RISC) optimized for Xilinx FPGAs [18]. Using the Xilinx Embedded Development Kit (EDK) simplifies the addition of peripherals, (e.g., I/O, memory, communications, etc). In order to test this design, eight 32-bit wide dual-port BRAMs were added to the Microblaze core to store the messages to be hashed as well as the results from the Skein core. A simple general purpose input/output (GPIO) interface was also added to connect the startCore, coreDone, mode, and numMessages signals between Microblaze and the Skein core. Lastly, a Universal Asynchronous Receiver Transmitter (UART) peripheral was added to provide a communication link between the ML605 development board and a PC for debugging and printing the test results. Figure 5.2 shows the high-level block diagram of the hardware testing framework.

Figure 5.2: Hardware testing framework

With the Microblaze core constructed, EDK generates the necessary C libraries with functions that allow access to the processor peripherals. A C program was written to run the tests similarly to the VHDL testbench. The program first stores the message(s) to be hashed in the embedded memory, as well as how many blocks each message contains, whether the message was padded, and how many total bytes were in the original message before padding. Once this process is complete, Microblaze asserts the correct mode and numMessages to the core and then asserts startCore. When hashing is complete, the resulting digests are written to the shared embedded memory, and the coreDone signal is asserted. Microblaze reads the results from the embedded memory and verifies them against the correct digests, printing any failed results to the PC terminal connected to the UART.

The basic flow of the hardware testing framework for both the Microblaze processor and the Skein core is shown in Figure 5.3.

Figure 5.3: Hardware testing flow

5.3 Synthesis and Implementation Automation

In an attempt to synthesize, implement, and test the proposed architectures in this work, a tool developed at George Mason University was utilized [5]. The Automated Tool for Hardware EvaluationN otherwise known as ATHENa is a set of scripts and configuration files that simplify the process of implementing, optimizing, and verifying the design. Currently ATHENa supports both Xilinx and Altera FPGAs.

The scripts use the embedded design tools provided by the user (e.g., Xilinx ISE, ModelSim, etc.) for synthesis, implementation, and functional verification (VHDL simulations). Two main configuration files must be edited by the user to specify how ATHENa will run. The first file must contain a list of all HDL source files or design files that are required to synthesize the design. The second configuration file specifies the location of source files, output directory location for reports and output design files, target devices, optimization method, etc. Formulas for the latency and throughput of a particular algorithm can also be entered in this configuration file, and ATHENa will calculate which design results in the best throughput, latency, and throughput per area. For this work, multiple target devices were specified in the configuration file so that ATHENa must run only once for all target devices, which include Virtex-5 XC5VLX110-3, Virtex-6 XC6VLX240T-1, and Virtex-6 XC6VLX240T-3. The key advantage to using the ATHENa tool is the optimization methods it provides, two of which are focused on. The optimization modes examined are Frequency Search and GMU Optimization as they are targeted at the frequency constraint of the synthesis and implementation tools. Additional optimization algorithms are provided for reduced area and custom exhaustive search algorithms. The Frequency Search algorithm is the more primitive of the two algorithms. The user provides a desired frequency for each target device in the configuration file. ATHENa then synthesizes and implements the design and determine if the desired frequency constraint was achieved. If the first implementation achieves this frequency, the frequency constraint for the next iteration will be increased by a percentage specified in the algorithm. Each time a frequency constraint is met, the constraint is increased. If the constraint is not met, the constraint is increased by a

lower percentage. This process is iterated until no frequency improvement is achieved. The GMU Optimization algorithm is more complicated and modifies many of the parameters of the tools to achieve better results. The algorithm not only modifies the frequency constraint but also the effort level of both the mapping and place and route stages. Multiple iterations with different cost tables are also used. Examples of performance improvements attained for the SHA-3 algorithms through the use of ATHENa are illustrated in Figure 5.4.

Figure 5.4: ATHENa results for SHA-3 candidates [8]

This work experimented with both the Frequency Search and GMU Optimization options of ATHENa and found that the GMU Optimization method produced the best results in terms of both maximum clock frequency and overall area for this architecture. Implementations were run for both Virtex-5 and Virtex-6 devices. The results are shown in Tables 5.1, 5.2, and 5.3 for the standard pipelined architecture (SP) and the deep-pipelined

architecture (DP). Improvement percentages are measured against implementations with no optimizations set in the tools.

Table 5.1: Skein-256 - Device : Virtex-5 XC5VLX110-3

Optimization Method	Design	Slices	CLK_{SEQ} (MHz)	CLK_{PIPE} (MHz)	Number of runs
Frequency Search	SP	3278	60.0	260.8	5
GMU_Optimization	SP	3251	61.1	264.3	62
Frequency Search	DP	3847	54.5	282.6	6
GMU_Optimization	DP	3895	58.3	291.2	65

Table 5.2: Skein-256 - Device : Virtex-6 XC6VLX240T-1

Optimization Method	Design	Slices	CLK_{SEQ} (MHz)	CLK_{PIPE} (MHz)	Number of runs
Frequency Search	SP	2526	48.9	228.3	7
GMU_Optimization	SP	2547	55.8	246.7	72
Frequency Search	DP	3018	49.1	252.4	5
GMU_Optimization	DP	2972	52.4	269.3	67

Table 5.3: Skein-256 - Device : Virtex-6 XC6VLX240T-3

Optimization Method	Design	Slices	CLK_{SEQ} (MHz)	CLK_{PIPE} (MHz)	Number of runs
Frequency Search	SP	2415	57.8	255.1	6
GMU_Optimization	SP	2516	71.1	301.9	57
Frequency Search	DP	3054	58.6	310.3	4
GMU_Optimization	DP	2989	67.7	330.5	61

It is important to note that the sequential frequencies were obtained directly from the post place and route reports, whereas the pipelined frequencies were obtained from reports generated by the timing analyzer. Timing was also verified in hardware for the Virtex-6 XC6VLX240-1 (ML605 Development Board). From these results, it is apparent that in most cases, ATHENa is able to achieve significant improvements in frequency with minimal increase in area and in some cases reduced area.

Chapter 6

Performance Results & Analysis

In an effort to compare the architecture designed in this work accurately and fairly to previous designs, equations must be developed to determine the latency and throughput for the different modes of hashing supported. The versatile Skein architecture was developed using many concepts from previous work, and thus some of the metrics used in [1] can be used and/or modified to measure the performance of the versatile Skein architecture.

Table 6.1: Variables for evaluating Skein hardware performance

Variable	Description
LAT_{SEQ}	Number of cycles to process single block of Threefish in sequential mode
LAT_{PIPE}	Number of cycles to process a single block of Threefish in multiple message or tree mode
CLK_{SEQ}	Maximum clock frequency of sequential data path
CLK_{PIPE}	Maximum clock frequency of pipelined data path
$msgBlocks_i$	Number of msgBlocks for a given message i
$numMsgs$	Number of messages to hash in multiple message mode
$pipelineRegs$	Number of pipeline registers (10 for standard pipeline, 20 for deep pipeline)
$OH_{STARTUP}$	Overhead cycles before start of hashing
OH_{OUTPUT}	Overhead cycles to complete output UBI of Skein
N_b	Number of bytes in a single message block
Y_L	Tree leaf-size encoding
Y_F	Tree fan-out encoding
Y_M	Maximum tree height
UBI_L	Total number of UBIs at the leaf level in a tree
UBI_F	Total number of UBIs at all levels except leaf level
LAT_L	Total number of cycles to process leaf level of tree
LAT_F	Total number of cycles to process all levels except leaf level

Before developing the equations to calculate the performance of this architecture for each of the modes, a set of variables is defined for these equations. Table 6.1 lists these

variable along with some previously defined variables for reference.

6.1 Sequential Hashing

Since the system clock frequency CLK_{PIPE} of the FPGA is used to drive all flip-flops in the design, when hashing a single message in sequential mode, the internal state register is not enabled every clock cycle. Enabling this register every $CYCLES_{EN}$ clock cycles creates a multicycle path. In order to determine $CYCLES_{EN}$ between enabling the internal state register, the delay of each of the independent paths in the Threefish core must be considered. From the delay of the sequential (unpipelined) path, the maximum clock frequency, CLK_{SEQ} , for that path can be determined. Similarly, the maximum clock frequency for the pipelined path, CLK_{PIPE} , can be determined. Thus the number of cycles between enable pulses when operating in sequential mode is defined by Equation 6.1. With $CYCLES_{EN}$ defined, the total latency of Threefish in sequential mode, LAT_{SEQ} , is defined by Equation 6.2. Subsequently this latency can be substituted in the throughput equation for sequential hashing in [1] resulting in Equation 6.3.

$$CYCLES_{EN} = \left\lceil \frac{CLK_{PIPE}}{CLK_{SEQ}} \right\rceil \quad (6.1)$$

$$LAT_{SEQ} = CYCLES_{EN} \cdot 10 \quad (6.2)$$

$$TP_{SEQ} = \frac{CLK_{SEQ} \cdot msgBlocks \cdot N_b \cdot 8}{OH_{STARTUP} + msgBlocks \cdot LAT_{SEQ} + OH_{OUTPUT}} \quad (6.3)$$

The $OH_{STARTUP}$ and OH_{OUTPUT} variables signify the number of cycles before the message hashing begins and the number of cycles to complete the output UBI(s) of Skein. Due to the storage of the number of blocks in a particular message in memory, sequential hashing requires a single cycle to retrieve this value before starting to hash the first message block. Thus, for sequential hashing $OH_{STARTUP} = 1$. For Skein variants where the

output size is equal to the state size as used in this work, the calculation of TP_{SEQ} uses $OH_{OUTPUT} = LAT_{SEQ}$.

6.2 Multiple Message Hashing

The pipelined Skein architecture was only researched in [9] and no equations have been developed to define accurately the performance of this hashing method. The throughput reported in [9] was the absolute maximum throughput of the pipelined core assuming all pipeline registers are always utilized. When hashing multiple messages, each message may not contain the same number of blocks, and there may not be enough messages to fill the pipeline completely. Both when the pipeline is completely filled and not completely filled must be addressed when calculating the throughput in multiple message mode. Using the number of messages, $numMsgs$, the number of blocks for each message, $msgBlocks_i$, and the number of pipeline stages in the Threefish core, $pipelineRegs$, the throughput, TP_{MM} , for a given set of messages is given by the summation in Equation 6.4. When $numMsgs = pipelineRegs$ and all the messages contain the same number of blocks, the maximum throughput is achieved and is given by Equation 6.5.

$$TP_{MM} = \sum_{i=1}^{pipelineRegs} \frac{CLK_{PIPE} \cdot msgBlocks_i \cdot N_b \cdot 8}{OH_{STARTUP} + (LAT_{PIPE} \cdot \max(msgBlocks)) + OH_{OUTPUT}} \quad (6.4)$$

$$\max(TP_{MM}) = \frac{CLK_{PIPE} \cdot msgBlocks \cdot N_b \cdot 8 \cdot pipelineRegs}{OH_{STARTUP} + (LAT_{PIPE} \cdot msgBlocks) + OH_{OUTPUT}} \quad (6.5)$$

Similarly to sequential hashing, additional cycles are required to read the number of blocks for each message from memory before beginning hashing. For multiple message hashing with the pipelined core, $OH_{STARTUP} = 10$, and for the deep pipelined core,

$OH_{STARTUP} = 20$. For Skein-256-256, $OH_{OUTPUT} = LAT_{PIPE}$ when hashing multiple messages.

Table 6.2: Skein 256 Eight-round unrolled results

Designer	Device	Slices	CLK_{SEQ} (MHz)	CLK_{PIPE} (MHz)	LAT_{SEQ} (Cycles)	LAT_{PIPE} (Cycles)	TP_{SEQ} (Mbps)	TP_{PIPE} (Mbps)
Schorr	XC5VLX110-3	1281	68.6	N/A	10	N/A	1755.5	N/A
Tillich	XC5VLX110-3	937	68.4	N/A	10	N/A	1751.0	N/A
ThisWork*	XC5VLX110-3	3251	61.1	264.3	50	102	1353.2	6633.4
ThisWork*	XC6VLX240T-1	2547	55.8	245.6	50	102	1257.5	6164.1
ThisWork*	XC6VLX240T-3	2516	71.1	301.9	50	102	1545.7	7577.1
ThisWork(DP)*	XC5VLX110-3	3895	58.3	291.2	50	202	1490.9	7380.9
ThisWork(DP)*	XC6VLX240T-1	2972	52.4	269.3	60	202	1149.0	6825.8
ThisWork(DP)*	XC6VLX240T-3	2989	67.7	330.5	50	202	1692.2	8377.0
*Implementation results optimized for pipelined modes								

In developing the versatile round architecture, the Xilinx dedicated carry-chain primitives were used. When synthesizing the design, these primitives are treated as black boxes, and the timing values used for the maximum clock frequency estimate are very inaccurate. Therefore, the implementation frequencies are the most accurate comparison. The sequential throughput TP_{SEQ} shows a decrease in performance due to the fact that the system frequency CLK_{PIPE} limits the ability of the architecture to run in sequential mode at the reported CLK_{SEQ} . Also, the reported CLK_{SEQ} from synthesis for previous designs are likely higher than they would be after implementation. However, the multiple message mode shows a significant increase in maximum throughput. A speedup of 4.9 is seen for each implementation as the ratio of latencies ($50/102$) times the number of messages processed in tree mode (10) is 4.9. Once again, this maximum speedup is achieved only when the pipeline is 100% utilized, (i.e. $numMsgs = pipelineRegs$ and all messages have the same number of $msgBlocks$). Although the pipelined architecture contains 10 “cores,” the increase in latency prevents the design from providing an ideal speedup of 10. Although this design was optimized to favor the multiple message and tree hashing, it could be equally feasible to optimize the design for sequential hashing. This optimization method would reduce the system clock speed CLK_{PIPE} so that it is an integer multiple closest but less than to the maximum achievable system clock speed. This allows for sequential hashing to run at its highest possible performance. The third and final way this design could be optimized is to implement dynamic clock frequencies. This would allow the hardware to access the FPGAs clocking resources and modify the system clock frequency when switching between hashing modes. In doing so, all modes would be optimized for their best possible performance. This is a suggested area of future work.

6.3 Pipelined Tree Hashing Performance

The most complex mode, the pipelined tree hashing mode, requires an in-depth analysis to determine how the tree parameters Y_L and Y_F affect throughput. A similar analysis was

conducted in [1] and [16] for the duplicate unrolled core architecture. This work modifies those results and applies them to the pipelined tree hashing mode. Many of the assumptions made in previous work are also removed in the development of the performance equations. The overall message overhead for a given message and different Y_L and Y_F parameters is also explored.

6.3.1 Message Processing Overhead

To better understand the effects of the tree hashing parameters Y_L and Y_F , one can examine the message overhead created in tree hashing versus sequential hashing. In sequential hashing, the total number of message blocks processed is equal to the number of message blocks in the message, and no message overhead is incurred. When tree hashing, each block of the message is processed at the leaf node level, and the results of the leaf level UBIs form a new message. The number of message blocks in each of these new messages formed at each tree level constitutes the message overhead for tree hashing. Examining the example trees in Figure 6.1, the message overhead in the first tree with $Y_L = 1$ and $Y_F = 1$, the tree must process 7 additional message blocks after the leaf level for a total of 15 message blocks. For the example with $Y_L = 1$ and $Y_F = 2$, the message overhead is decreased to 4 message blocks.

Figure 6.1: Tree structure with different Y_L and Y_F [1]

Thus, an additional message block must be processed for each UBI in the tree except for the root node. Therefore the message overhead for a given tree is given by Equation 6.6 where the number of UBIs for a given message is calculated using the algorithm from [16] shown in Table 6.3 where UBI_L represents the number of UBIs at the leaf level of the tree and UBI_F represents the remaining UBIs at all other levels of the tree. The minimum possible overhead for $Y_L = Y_F$ is given by Equation 6.7 where k is the resulting number of tree levels for a given message.

$$MSG_{OH} = \frac{UBI_L + UBI_F - 1}{msgBlocks} \quad (6.6)$$

$$\lim_{k \rightarrow +\infty} MSG_{OH} = \sum_{i=1}^k \frac{1}{2^{Y_F^k}} = \frac{1}{2^{Y_F} - 1} \quad (6.7)$$

As Y_L and Y_F are increased, the number of messages blocks processed by each UBI is increased, and therefore fewer UBIs are needed for a message of a given size. In Figure

UBI Nodes in Tree Algorithm
$UBI_L = \lceil \frac{msgBlocks}{2^{Y_L}} \rceil$ $UBIs = UBI_L$
<pre> while(UBIs > 1) { $UBIs = \lceil \frac{UBIs}{2^{Y_F}} \rceil$ $UBI_F = UBI_F + UBI_s$ } </pre>

Table 6.3: Calculating the UBI nodes in a Skein tree [1]

6.2, the overhead for a message of 8192 blocks with $Y_F = Y_L$ is illustrated. As one would expect, as Y_F and Y_L are increased the number of UBIs in the tree decrease and thus less additional messages need to be processed.

Figure 6.2: Message overhead for $Y_F = Y_L, msgBlocks = 8192$

In order to better understand the impact of Y_F and Y_L separately, Figures 6.3(a) and 6.3(b) analyze each of the parameters independently. In Figure 6.3(a), Y_L is held constant at 1, and Y_F is varied. It is evident that as Y_F increases, the message overhead decreases. However, when compared to Figure 6.3(b), where Y_L is varied and Y_F is fixed at 1, the decreased is not a drastic. It can be concluded that the reduction of the original message at the leaf level is the dominant factor in determining overall message overhead.

(a) Message overhead for varying Y_F , $msgBlocks = 8192$

(b) Message overhead for varying Y_L , $msgBlocks = 8192$

Figure 6.3: Message overhead for tree processing

The graphs in Figure 6.4 illustrate the message overhead as a percentage of the original message size for Y_L and Y_F values of 1, 2, 4, and 8. It is clear that as each parameter is increased the maximum message overhead as well as the asymptote of the message overhead decreases. The message overhead is lowest when $Y_L = 8$ and $Y_F = 8$ at $1/255$ or 0.39%.

Figure 6.4: Message overhead for tree hashing

6.3.2 Tree Performance

To measure the performance of Skein’s tree hashing mode, the resulting tree structures for given messages must be carefully analyzed. Previous equations developed in [1] and [16] made particular assumptions regarding the utilization of multiple cores. In particular, regardless of the actual tree structure, it was assumed in [16] that all cores are fully utilized at all times for any size message and that the resulting tree structure for particular message is “complete”. For a tree to be considered complete, each UBI at the leaf level must process exactly 2^{Y_L} message blocks, and each UBI at the remaining levels must process exactly 2^{Y_F} message blocks. It is obvious that only for messages of certain sizes for a given Y_F and Y_L will a tree be complete. This work has developed an algorithm to better describe the performance of tree hashing by removing these assumptions.

The following algorithms in Tables 6.4 and 6.5 calculate the latency for both the leaf level UBIs and remaining UBIs separately as LAT_L and LAT_F respectively. Subsequently, the same equation from [16] shown in Equation 6.8 can be used to determine the throughput. These algorithms are much more accurate as they determine if there are UBIs that do not process either exactly 2^{Y_L} message blocks for leaf level UBIs or exactly 2^{Y_F} message blocks for subsequent levels of the tree. The separate algorithms also help in showing the different impacts of Y_L and Y_F .

$$TP_{TREE} = \frac{CLK_{PIPE} \cdot msgBlocks \cdot N_b \cdot 8}{OH_{STARTUP} + (LAT_L + LAT_F) + OH_{OUTPUT}} \quad (6.8)$$

Table 6.4: Calculation of LAT_L

```

 $LAT_L = 0$ 
 $UBI_L = \lceil \frac{msgBlocks}{2^{Y_L}} \rceil$ 
if( $UBI_L > pipelineRegs$ )then
     $LAT_L = 2^{Y_F} \cdot \left\lfloor \frac{UBI_L}{pipelineRegs} \right\rfloor$ 
     $remainder = UBI_L - \left( pipelineRegs \cdot \left\lfloor \frac{UBI_L}{pipelineRegs} \right\rfloor \right)$ 
    if( $remainder == 1$ )then
         $LAT_L = LAT_L + (msgBlocks - (2^{Y_L} \cdot (UBI_L - 1)))$ 
    else
         $LAT_L = LAT_L + 2^{Y_L}$ 
    end if
else if( $UBI_L == 1$ )then
     $LAT_L == msgBlocks$ 
else
     $LAT_L = 2^{Y_L}$ 
end if
 $LAT_L = LAT_L \cdot LAT_{PIPE}$ 

```

Table 6.5: Calculation of LAT_F

```

done = 0
LAT_F = 0
UBI_PREVLEVEL = UBI_L
while(done == 0)do
  UBI_F =  $\lceil \frac{UBI\_PREVLEVEL}{2^{Y_F}} \rceil$ 
  if(UBI_F > pipelineRegs)then
    LAT_F = LAT_F +  $\left( 2^{Y_F} \cdot \left\lfloor \frac{UBI_F}{pipelineRegs} \right\rfloor \right)$ 
    remainder = UBI_F -  $\left( pipelineRegs \cdot \left\lfloor \frac{UBI_F}{pipelineRegs} \right\rfloor \right)$ 
    if(remainder == 1)then
      LAT_F = LAT_F + (UBI_PREVLEVEL -  $(2^{Y_F} \cdot (UBI_F - 1))$ )
    else
      LAT_F = LAT_F +  $2^{Y_F}$ 
    end if
  else if(UBI_F == 1)then
    LAT_F = LAT_F + UBI_PREVLEVEL
    done = 0
  else
    LAT_F = LAT_F +  $2^{Y_F}$ 
  end if
  UBI_PREVLEVEL = UBI_F
end while
LAT_F = LAT_F · LAT_PIPE

```

As an example, consider the proposed tree hashing situation from Figure 6.5 that was shown earlier where $Y_L = Y_F = 4$ and $msgBlocks = 4096$.

Figure 6.5: $Y_L = 4$, $Y_F = 4$, $msgBlocks = 4096$

Using the formulas above, the latency of the leaf level nodes LAT_L is $26 \cdot LAT_{PIPE} \cdot 2^{Y_L}$ because a total of 256 UBIs are needed to process the leaf level. The remainder of the tree is comprised of 17 UBIs including the root node. Since the root node requires the results from all below it, it cannot run simultaneously with any other UBIs. During the last stage of leaf level processing 4 pipeline stages are left open, these can be used to process the second level of the tree. Account for these two factors the latency of the remainder of the tree is $3 \cdot LAT_{PIPE} \cdot 2^{Y_F}$ for a total of $29 \cdot LAT_{PIPE} \cdot 2^{Y_F}$ or $464 \cdot LAT_{PIPE}$ which correlates with Figure 6.5.

The graphs shown in Figures 6.6 and 6.7 illustrate the speedup achieved for different tree parameters. As stated earlier, the maximum speedup for the standard pipelined core is 4.9. From the discussion of message overhead, the greater the value of Y_F and Y_L , the lower the overhead and thus one would expect increased throughput. In Figure 6.7, where $Y_F = Y_L$, the results do indicate that the greater the parameters the greater the speedup. However, $Y_F = Y_L = 8$ does not yield the best speedup of until the messages become much larger. The slow growth of the speedup of $Y_F = Y_L = 8$ is shown in 6.8. Examining the graphs in Figure 6.6, it is evident that the greatest speedups are achieved when $Y_L = 8$,

but not while $Y_F = 8$ for smaller messages. With the lower values of Y_F , a greater number of UBIs is created but each with a decreased total latency.

Figure 6.6: Maximum theoretical speedup of pipelined tree hashing

The large number of pipeline stages is capable of processing a greater number of UBIs each with a decreased latency quicker than few UBIs with longer latencies as is the case with $Y_F = 8$. As the message size increases greatly, the number of UBIs increases even when $Y_F = Y_L = 8$, and will approach the maximum speedup. When fewer UBIs exist on a level, more pipeline stages are wasted. For instance, when $Y_F = Y_L = 8$ and $256 \leq msgBlocks \leq 65536$, only two levels exist in the tree with the second level being the root node. Once the root node is reached in the processing, all but one pipeline stage is wasted until processing is complete. Efficiency graphs are also provided in Figures 6.9 and 6.10. These graphs follow from the speedup graphs and represent the ratio of achieved speedup

versus the maximum possible speedup for the pipelined core. The values for maximum achievable throughput are shown in Table 6.6. These values are equivalent to those of the multiple message hashing as the ideal tree hashing would be able to fully utilize all pipeline stages except when processing the root node. For very large messages, the time to process the root node would be negligible in comparison to processing the rest of the tree.

Figure 6.7: Maximum theoretical speedup of pipelined tree hashing $Y_F = Y_L$

Figure 6.8: Maximum theoretical speedup of pipelined tree hashing $Y_F = Y_L$

Figure 6.9: Maximum theoretical efficiency of pipelined tree hashing

Figure 6.10: Maximum theoretical efficiency of pipelined tree hashing $Y_F = Y_L$

Table 6.6: Tree hashing results

Designer	Device	Slices	Clk (MHz)	TP_{TREE} (Mbps)	Num Cores
Schorr [16]	XC5VLX110-3	3024	59.3	3037.0	2
Schorr [16]	XC5VLX155-3	5657.0	54.9	5625	4
ThisWork	XC5VLX110-3	3251	264.3	6633.4	1
ThisWork	XC6VLX240T-1	2547	245.6	6164.1	1
ThisWork	XC6VLX240T-3	2516	301.9	7577.1	1

Although for most messages tree hashing provides much greater performance, smaller message can be hashed faster in sequential mode for certain cases. An area of future work may be to intelligently determine which mode to use based on message size, if sequential mode is not required. In order to do so, the cutoff point where tree mode surpasses sequential mode in throughput must be analyzed. The graphs in Figures 6.11 and 6.12 compare the sequential mode execution time to the tree hashing execution time, only for the message processing stage. It is obvious that messages that only require one UBI in tree mode will be worse than the execution time of sequential mode as the latency of the pipeline used for tree mode is greater than that of the unpipelined data path used in sequential mode. This is confirmed by the graphs. The worst case for the tree mode compared to sequential mode is when $Y_L = 8$. Tree mode does not outperform sequential mode in this case until between 500–550 depending upon the value of Y_F . These are small messages, and it is safe to say that tree hashing outperforms sequential hashing in essentially all cases.

Figure 6.11: Theoretical execution time

81

Figure 6.12: Theoretical execution time $Y_F = Y_L$

Chapter 7

Conclusions

This work accomplished several goals. A versatile FPGA architecture for Skein was developed. This architecture is capable of hashing messages in three different modes, sequential, multiple message, and pipelined tree hashing. This architecture used FPGA resources efficiently so that no CLB overhead was introduced in the Threefish function. An additional implementation explored the use of a “deep pipeline” that split 64-bit additions into two 32-bit addition executed in two clock cycles in order to reduce the critical path. The ATHENA automation tools were used to automate the synthesis and implementation process as well as optimize the final hardware implementation. The fully autonomous architecture was verified both in simulation and in hardware using a custom testing framework that utilized the Microblaze soft-processor.

Comprehensive analysis of the performance of this architecture with respect to each operating mode was conducted using equations developed in this work or modified from previous work. From this analysis the benefits of having each of the modes are clear. Single messages are hashed more efficiently in sequential mode as the latency is decreased. The pipelined tree hashing mode can also be used to speed up the hashing of a single message, especially for larger messages where the message overhead is decreased. For hashing multiple unique messages simultaneously, the pipelined architecture greatly increases the throughput.

There are a few areas of this architecture that should be explored in future work. The versatile Skein architecture in this work was able to utilize the hardware elements of Xilinx

FPGAs efficiently. However, this architecture may be modified to accommodate the different hardware elements of FPGAs from other vendors such as Altera. The Skein-512-224, Skein-512-256, Skein-512-384 and Skein-512-512 variants should also be explored as they are recommended for SHA-3.

The second area of future work to be explored is duplicating the pipelined core similarly to the way that the standard unrolled core was duplicated in [1]. The difficulty in duplicating the unrolled and pipelined core would be the message delivery. This work assumed an ideal memory interface where an entire message block can be delivered each clock cycle. As multiple pipelined cores are added, the memory requirement would be significant. Since the latency of the pipelined core is much longer than that of the standard unrolled core, the additional cycles may be used to read the messages from memory into a buffer before they are needed. The performance of architectures with both a single or multiple pipelined cores when using non-ideal memories should also be conducted in future work.

The pipelined architecture developed in this work is capable of hashing multiple messages simultaneously or a single message in tree mode. If a modification to the Skein algorithm provided a mode where entire blocks of a single message could be streamed in at a time, without having to wait for previous blocks to complete, the message overhead created by the tree mode would be eliminated. The security of such a modification would have to be carefully examined by the developers of Skein.

Bibliography

- [1] Aric Schorr. Performance Analysis of a Scalable Hardware FPGA Skein Implementation. Master's thesis, Rochester Institute of Technology, Rochester, New York, February 2010.
- [2] Brian Baldwin, Neil Hanley, Mark Hamilton, Liang Lu, Andrew Byrne, Maire O'Neill, and Willaim P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. Available online at <http://www.ucc.ie/en/crypto/SHA-3Hardware/NISTSHA-3/Baldwin-SHA-3-Paper-Aug-2010>, August 2010.
- [3] Cameron McDonald, Philip Hawkes, and Josef Pieprzyk. Differential Path for SHA-1 with complexity $O(2^{52})$. Cryptology ePrint Archive (<http://eprint.iacr.org/>), Report 2009/256, June 2009.
- [4] Computer Security Division. National Institute of Standards and Technology (NIST). Cryptographic Hash Project Website at <http://csrc.nist.gov/groups/ST/hash/index.html>.
- [5] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Usings FPGAs. Cryptology ePrint Archive (<http://eprint.iacr.org/>), Report 2010/445, December 2010.
- [6] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs. Available online at <http://www.ecrypt.eu.org/hash2011/proceedings/>, May 2011.
- [7] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, 2003.
- [8] Ekawat Homsirikamol, Marcin Rogawski, Kris Gaj, Jens-Peter Kaps, Venkata Amirineni, and Benjamin Y. Brewster. ATHENa - Automated Tool for

- Hardware Evaluation : Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware using FPGAs. PowerPoint Presentation. Available online at http://cryptography.gmu.edu/athena/presentations/FPL_2010_gaj.pdf, 2010.
- [9] Jesse Walker, Farhana Sheikh, Sanu K. Mathew, and Ram Krishnamurthy. A Skein-512 Hardware Implementation. Available online at http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/WALKER_skein-intel-hwd.pdf.
 - [10] Kazuyuki Kobayashi, Jun Ikegami, and Shin'ichiro Matsuo, Kazuo Sakiyama, and Kazuo Ohta. Evaluation of Hardware Performance for the SHA-3 Candidates Using SASEBO-GII. Cryptology ePrint Archive (<http://eprint.iacr.org/>), Report 2010/010.
 - [11] Men Long. Implementing Skein Hash Function on Xilinx Virtex-5 FPGA Platform. Available online at http://www.skein-hash.info/sites/default/files/skein_fpga.pdf, February 2009.
 - [12] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.
 - [13] Meltem Sonmez Turan, Ray Perlner, Lawrence E. Bassham, William Burr, Don-hoon Chang, Morris J. Dworkin, Shu-jen Chang, John M. Kelsey, Rene Peralta, and Souradyuti Paul. Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition. Technical report, National Institute of Standards and Technology, February 2011.
 - [14] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family v1.3. Available online at <http://www.skein-hash.info/sites/default/files/skein1.3.pdf>, October 2010.
 - [15] Richard F. Kayser. Announcing request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 72(212), November 2, 2007.
 - [16] Aric Schorr and Marcin Lukowiak. Skein tree hashing on fpga. In *Reconfigurable*

Computing and FPGAs (ReConFig), 2010 International Conference on, pages 292–297, dec. 2010.

- [17] Stefan Tillich. Hardware Implementation of the SHA-3 Candidate Skein. Cryptology ePrint Archive (<http://eprint.iacr.org/>), Report 2009/159, April 2009.
- [18] Xilinx. MicroBlaze Processor Reference Guide, UG081 (v12.0), March 2011.
- [19] Xilinx. Virtex-5 FPGA User Guide, UG190 (v5.3), May 2010.
- [20] Xilinx. Virtex-6 FPGA Configurable Logic Block User Guide, UG364 (v1.1), September 2009.