5-1-2008

FPGA design and performance analysis of SHA-512, whirlpool and PHASH hashing functions

Przemyslaw Zalewski

FPGA design and performance analysis of SHA-512, Whirlpool and PHASH hashing functions

by

Przemysław Zalewski

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

Supervised by

Dr. Marcin Łukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2008

Approved By:

Dr. Marcin Łukowiak
Assistant Professor, RIT Department of Computer Engineering
Primary Advisor

Dr. Stanisław P. Radziszowski
Professor, RIT Department of Computer Science

Dr. Kenneth Hsu
Professor, RIT Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: FPGA design and performance analysis of SHA-512, Whirlpool and PHASH hashing functions

I, Przemysław Zalewski, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Przemysław Zalewski

Date

Dedication

This work is dedicated to my parents, who have always been there for me through the good and the bad.

Acknowledgments

I would like to thank Dr. Marcin Łukowiak and Dr. Stanisław Radziszowski for all of their guidance and support throughout the duration of this work.

Abstract

Hashing functions play a fundamental role in modern cryptography. Such functions process data of finite length and produce a small fixed size output referred to as a digest or hash. Typical applications of these functions include data integrity verification and message authentication schemes. With the ever increasing amounts of data that needs to be hashed, the throughput of hashing functions becomes an important factor.

This work presents and compares high performance FPGA implementations of SHA-512, Whirlpool and a recently proposed parallelizable hash function, PHASH. The novelty of PHASH is that it is able to process multiple data blocks at once, making it suitable for achieving ultra high-performance. It utilizes the W cipher, as described in Whirlpool, at its core. The SHA (SHA-0, SHA-1, SHA-2) family of functions is one of the first widely accepted standards for hashing. According to currently published literature, the fastest SHA-512 (a variant of SHA-2) implementation achieves a throughput of 1550 Mbps. A recently introduced hashing function, Whirlpool, provides comparable security to SHA-512 and is able to achieve much better performance. According to currently published literature, the fastest Whirlpool implementation achieves a throughput of 4896 Mbps. The proposed PHASH hash function greatly outperforms both SHA-512 and Whirlpool.

All implementations are targeted for the state-of-the-art Xilinx Virtex-5 LX330 FPGA. The SHA-512 implementation attains a throughput of 1828 Mbps, and Whirlpool attains 7687 Mbps. PHASH achieves a throughput over 15 Gbps using a single W cipher instance. Using 8 W cipher instances a throughput over 100 Gbps is achieved and 16 instances provide a throughput over 182 Gbps.

Contents

List of Figures

List of Tables

Glossary

AES Advanced Encryption Standard.

BSB Base System Builder.

CF Compact Flash.

CLA Carry Look-ahead Adder.

CLB Configurable Logic Block.

CPA Carry Propagate Adder.

CSA Carry Save Adder.

DCM Digital Clock Manager.

DDR Double Data Rate.

EDK Embedded Development Kit.

FIFO First In First Out.

FIPS Federal Information Processing Standards.

FPGA Field Programmable Gate Array.

GF Galois Field.

IEC International Electrotechnical Commission.

IP Intellectual Property.

ISO International Organization for Standardization.

JTAG Joint Test Action Group.

LUT Look Up Table.

MDS Maximum Distance Separable.

MSB Most Significant Bit.

NESSIE New European Schemes for Signatures, Integrity and Encryption.

ODT On Die Termination.

PAR Place And Route.

PC Parallel Counter.

RAM Random Access Memory.

ROM Read Only Memory.

S-box Substitution box.

SATA Serial Advanced Technology Attachment.

SHA Secure Hash Algorithm.

UART Universal Asynchronous Receiver / Transmitter.

VHDL VHSIC Hardware Description Language.

Chapter 1

Introduction

Cryptographic hash functions are related to general hash functions, however they exhibit additional properties that general hash functions do not necessarily have. Such functions map a finite input into a small fixed size output, sometimes referred to as a digest. A good hashing function has pre-image resistance, 2^{nd} pre-image resistance and collision resistance. Pre-image resistance means that it should be computationally infeasible to find any input which produces a digest equal to a pre-specified output. 2^{nd} pre-image resistance means that it should be computationally infeasible to find any second input which has the same output as any specified input. Collision resistance means that it should be computationally infeasible to find two different inputs which hash to the same output.

Hashing functions can be divided into two categories: keyed and un-keyed. Keyed hash functions are used in message authentication schemes. Such functions take as an input, not only the data to be hashed, but also a 2^{nd} parameter, referred to as the key. The hashing algorithm uses the actual data along with the provided key to generate a hash.

Un-keyed functions are generally used to verify data integrity. The original data is passed through the function and the resulting digest is saved. At a later time a copy of the data is hashed and the two hashes are compared. If they are the same, for all intents and purposes, the original data and the copy are identical. However, if they differ, the integrity of the copy is not to be trusted, as the data may have been altered as a result of transmission errors, or even worse, it may have been altered by an outside source for malicious purposes. A more in-depth description and analysis of hashing functions can be found in [15].

Hashing functions are computationally intensive. Due to the iterative nature of such functions, implementations of these algorithms cannot take advantage of techniques such as parallelism in order to improve performance. General purpose processors are not well suited to efficiently execute the fundamental operations required for a given hashing algorithm. In order to increase performance hardware implementations of such algorithms are developed.

With the emergence of new hashing functions there must exist a way to compare and contrast them. It is important to differentiate between software and hardware implementations when performing such comparisons.

Software implementations, executed on general purpose processors, are used in many authentication protocols as well as in standalone applications. Hardware implementations, created using specialized circuitry, are often used in devices where it is impractical or impossible to use a software implementation, such as embedded systems or high speed networking devices. Execution time and memory usage are two metrics that can be used to compare software implementations. The lower the execution time and memory usage the more favorable a hashing function is. In hardware, execution time as well as overall area are two important metrics that need to be considered. For such implementations it is often the case that execution time can be decreased at the cost of increased overall area. The area can be calculated in terms of the physical size the resulting implementation occupies as well as by the number of logic gates used in order to realize it.

Hardware implementations can be used in order to achieve a much lower execution time than can be provided by software. Hardware devices such as FPGAs provide highly optimized operations that hashing algorithms require. As a result such implementations typically outperform software by several orders of magnitude.

Chapter 2

SHA-512, Whirlpool and PHASH

All of the hashing functions investigated in this work produce 512 bit hash values. SHA-512 and PHASH operate on 1024 bit chunks, or blocks, of data. Whirlpool performs computations on 512 bit blocks of data at one time.

2.1 SHA-512

SHA-512 is one of the five variants belonging to the SHA family of hashing functions. The other functions include SHA-1, SHA-224, SHA-256 and SHA-384. In August 2002, the SHA family, excluding SHA-224, was announced as a new standard as defined in the FIPS PUBS 180-2. In February 2004, a change to FIPS PUBS 180-2 included SHA-224 in the standard.

SHA-512 operates on messages less than 2^{128} bits or over 10^{25} terabytes, in length. It is an iterative algorithm consisting of four stages: message padding, message expansion, message compression and the hash update. In order to process a single block of data 80 iterations, or rounds, of the message compression stage are performed. Figure 2.1 shows a high level block diagram of the stages involved in the SHA-512 hashing algorithm. A more detailed diagram is shown in Figure 2.2.

The notation shown in Table 2.1 will be used to facilitate the description of the SHA-512 hashing algorithm. The SHA-512 algorithm requires evaluation of six logical functions every iteration. These functions consist of simple operations such as shift, rotate and

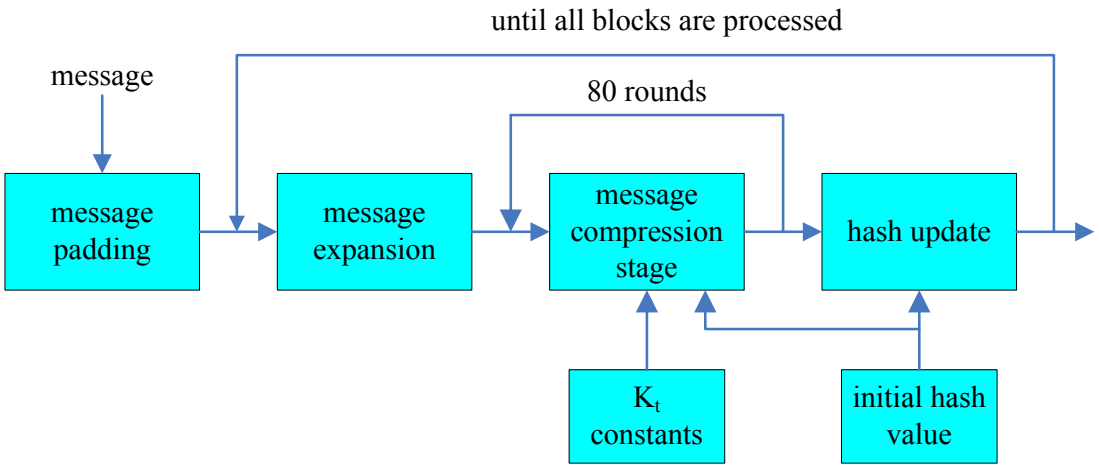

Figure 2.1: *High level block diagram depicting the structure of the SHA-512 hashing algorithm*

exclusive-or, referred to as $\oplus$ throughout the rest of this work. The function definitions are shown in Table 2.2.

The algorithm also requires two sets of constants: the initial hash value and a matrix, K_t, consisting of the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. The initial hash value constants are shown in Table 2.3. Table 2.4 contains the K_t constants. The values are oriented in a left to right, top to bottom fashion to save space. The K_t constants are used in the message compression stage of the algorithm.

2.1.1 Message padding

In order to ensure that the data to be hashed is aligned on the appropriate bit boundary a padding stage, also known as Merkle-Damgård strengthening is performed on the data to be hashed. First a single '1' bit is appended to the end of the message, followed by a variable number of '0' bits. The number of '0' bits appended is such that the message length becomes a multiple of 896. Then the length of the original message, before any padding has been done, is appended as an unsigned 128 bit integer. As a result, padding

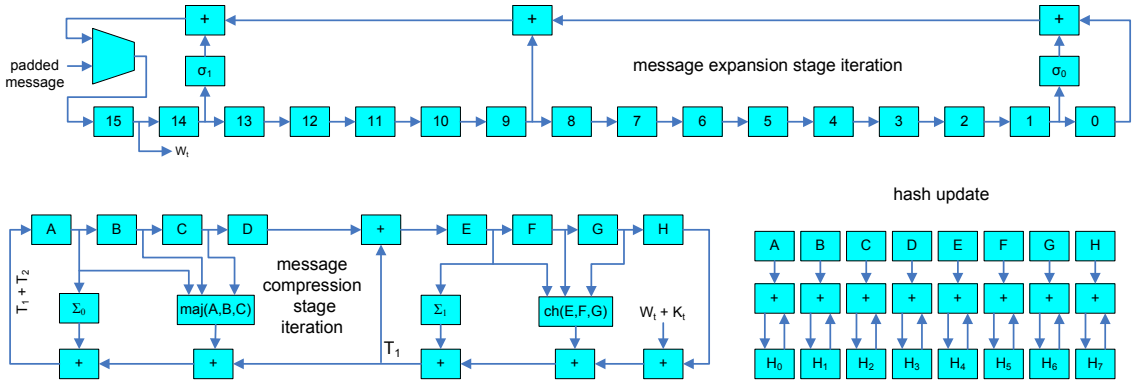

Figure 2.2: *Detailed block diagram depicting the structure of the SHA-512 hashing algorithm [13]*

produces a message that is an exact multiple of 1024 bits.

2.1.2 Message expansion

The message expansion stage generates the message schedule by performing a series of operations on the data contained in the current 512-bit block as shown in Table 2.5. This results in a message schedule consisting of 5120 bits of data, since each W_t is 64 bits long, and there are 80 such W_t's, one for each iteration of the message schedule. It should be noted that the first 1024 bits of the message schedule consists simply of the data block currently being operated on.

2.1.3 Message compression

The message compression stage takes the 5120 bits generated by the message schedule and compresses them to 512 bits. These bits are used to represent the hash value of the current block. During the first iteration, working variables a through h are initialized with the values shown in Table 2.3. Just like the message expansion stage, this stage consists of 80 iterations. For each iteration, t, the sequence of operations performed is shown in Table 2.6. After 80 iterations are completed the values of a through h are used in the hash update stage.

a though h	64 bit working variables used in the computation of hash values
H_j^i	The j^{th} 64-bit word of the i^{th} hash value
K_t	Constant used in the t^{th} iteration of the message compression stage
M_j^i	The j^{th} 64-bit word of the i^{th} block of the original message to be hashed
N	Number of blocks in the padded message
T_1 , T_2	Temporary variables used in the message compression stage
W_t	The t^{th} 64-bit word generated by the message schedule
$+$	Addition modulo 2^{64}
$\vee$	Logical OR operation
$\wedge$	Logical AND operation
$\oplus$	Exclusive-or operation
$\neg$	Bitwise complement operation
$ROTR^n(x)$	Circular shift right of x by n positions
$SHR^n(x)$	Shift right of x by n positions. Values shifted in are '0' bits

Table 2.1: *Notation used to facilitate the description of the SHA-512 hashing algorithm [7]*

2.1.4 Hash update

The initial hash value represented by H_j^0 is initialized using the constants shown in Table 2.3. The hash update stage simply accumulates the final values of a through h after the message compression completes. In other words, the hash update stage executes once for each data block of the padded message. The accumulated values are stored in the H_j^i variables as shown in Table 2.7. The final hash value consists of the concatenation of H_0^N through H_7^N.

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \\
\Sigma_1(x) &= ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \\
\sigma_0(x) &= ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \\
\sigma_1(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)
\end{aligned}
$$

Table 2.2: *Definitions of logic functions used in the SHA-512 hashing algorithm [7]*

$$
\begin{aligned}
a = H_0^0 &= \texttt{6A09E667F3BCC908}_x \\
b = H_1^0 &= \texttt{BB67AE8584CAA73B}_x \\
c = H_2^0 &= \texttt{3C6EF372FE94F82B}_x \\
d = H_3^0 &= \texttt{A54FF53A5F1D36F1}_x \\
e = H_4^0 &= \texttt{510E527FADE682D1}_x \\
f = H_5^0 &= \texttt{9B05688C2B3E6C1F}_x \\
g = H_6^0 &= \texttt{1F83D9ABFB41BD6B}_x \\
h = H_7^0 &= \texttt{5BE0CD19137E2179}_x
\end{aligned}
$$

Table 2.3: *Initial hash value constants [7]*

2.2 Whirlpool

Whirlpool is a hashing function developed by Vincent Rijmen and Paulo Barreto [4]. It was one of the two hashing functions chosen, the other being the SHA-2 family of hashing functions, as part of the NESSIE initiative [16]. The NESSIE project called for new hashing algorithms to be developed and put through a rigorous 3 year evaluation and cryptographic analysis. The Whirlpool hashing algorithm has also been included in the ISO/IEC 10118-3 standard [8].

Whirlpool operates on messages less than 2^{256} bits or over 10^{64} terabytes, in length.

$$K_t =$$

$$\begin{bmatrix}
428A2F98D728AE22_x & 7137449123EF65CD_x & B5C0FBCFEC4D3B2F_x & E9B5DBA58189DBBC_x \\
3956C25BF348B538_x & 59F111F1B605D019_x & 923F82A4AF194F9B_x & AB1C5ED5DA6D8118_x \\
D807AA98A3030242_x & 12835B0145706FBE_x & 243185BE4EE4B28C_x & 550C7DC3D5FFB4E2_x \\
72BE5D74F27B896F_x & 80DEB1FE3B1696B1_x & 9BDC06A725C71235_x & C19BF174CF692694_x \\
E49B69C19EF14AD2_x & EFBE4786384F25E3_x & 0FC19DC68B8CD5B5_x & 240CA1CC77AC9C65_x \\
2DE92C6F592B0275_x & 4A7484AA6EA6E483_x & 5CB0A9DCBD41FBD4_x & 76F988DA831153B5_x \\
983E5152EE66DFAB_x & A831C66D2DB43210_x & B00327C898FB213F_x & BF597FC7BEEF0EE4_x \\
C6E00BF33DA88FC2_x & D5A79147930AA725_x & 06CA6351E003826F_x & 142929670A0E6E70_x \\
27B70A8546D22FFC_x & 2E1B21385C26C926_x & 4D2C6DFC5AC42AED_x & 53380D139D95B3DF_x \\
650A73548BAF63DE_x & 766A0ABB3C77B2A8_x & 81C2C92E47EDAEE6_x & 92722C851482353B_x \\
A2BFE8A14CF10364_x & A81A664BBC423001_x & C24B8B70D0F89791_x & C76C51A30654BE30_x \\
D192E819D6EF5218_x & D69906245565A910_x & F40E35855771202A_x & 106AA07032BBD1B8_x \\
19A4C116B8D2D0C8_x & 1E376C085141AB53_x & 2748774CDF8EEB99_x & 34B0BCB5E19B48A8_x \\
391C0CB3C5C95A63_x & 4ED8AA4AE3418ACB_x & 5B9CCA4F7763E373_x & 682E6FF3D6B2B8A3_x \\
748F82EE5DEFB2FC_x & 78A5636F43172F60_x & 84C87814A1F0AB72_x & 8CC702081A6439EC_x \\
90BEFFFA23631E28_x & A4506CEBDE82BDE9_x & BEF9A3F7B2C67915_x & C67178F2E372532B_x \\
CA273ECEEA26619C_x & D186B8C721C0C207_x & EADA7DD6CDE0EB1E_x & F57D4F7FEE6ED178_x \\
06F067AA72176FBA_x & 0A637DC5A2C898A6_x & 113F9804BEF90DAE_x & 1B710B35131C471B_x \\
28DB77F523047D84_x & 32CAAB7B40C72493_x & 3C9EBE0A15C9BEBC_x & 431D67C49C100D4C_x \\
4CC5D4BECB3E42B6_x & 597F299CFC657E2A_x & 5FCB6FAB3AD6FAEC_x & 6C44198C4A475817_x
\end{bmatrix}$$

Table 2.4: *K_t constants used in the message compression stage [7]*

$$W_t = \begin{cases} M_i^t & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 79 \end{cases}$$

Table 2.5: *Operations performed in the message expansion stage [7]*

It uses the Miyaguchi-Preneel compression scheme with a dedicated 512-bit block cipher called W. This block cipher is strongly based on the structure of the AES [17]. Whirlpool operates on the hash state using a chained key state. The hash state is computed based on the input data to be hashed, whereas the key state is computed based on an initialization vector. This vector consists of all '0' bits.

Figure 2.3 shows a block diagram of the Miyaguchi-Preneel compression scheme. Such a scheme requires two inputs, m_i and h_{i-1}, where m_i is the single block of data to be compressed and h_{i-1}, called the chaining variable, is the output of the previous iteration

$$
\begin{aligned}
T_1 &= h + \Sigma_1(e) + Ch(e,f,g) + K_t + W_t \\
T_2 &= \sigma_0(a) + Maj(a,b,c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned}
$$

Table 2.6: *Operations performed in the message compression stage [7]*

$$
\begin{aligned}
H_0^i &= H_0^{i-1} + a \\
H_1^i &= H_1^{i-1} + b \\
H_2^i &= H_2^{i-1} + c \\
H_3^i &= H_3^{i-1} + d \\
H_4^i &= H_4^{i-1} + e \\
H_5^i &= H_5^{i-1} + f \\
H_6^i &= H_6^{i-1} + g \\
H_7^i &= H_7^{i-1} + h
\end{aligned}
$$

Table 2.7: *Operations performed in the hash update stage [7]*

of the Miyaguchi-Preneel compression scheme. The output of the current iteration h_i, is dependent on m_i, the block of data currently being processed, h_{i-1}, the output of the previous iteration and w_i, the output of the W block cipher.

2.2.1 Message padding

In order to ensure that the data to be hashed is aligned on the appropriate bit boundary a padding stage is performed on the data to be hashed. First a single '1' bit is appended to the end of the message, followed by a variable number of '0' bits. The number of '0' bits appended is such that the message length becomes a multiple of 256. Then the length of the original message, before any padding has been done, is appended as an unsigned 256 bit integer. As a result, padding produces a message that is an exact multiple of 512 bits.

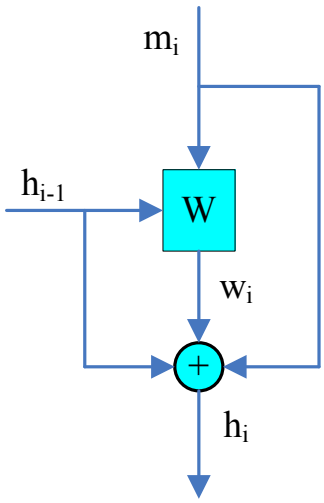

Figure 2.3: *Miyaguchi-Preneel compression scheme*

2.2.2 Operation

Once the message has been padded it is ready to be processed. A block of data is organized as an 8x8 array of bytes. In order to process a single block of data, 10 iterations, or rounds, of the W cipher are applied. After 10 rounds, the Miyaguchi-Preneel compression scheme is performed. This sequence repeats until all blocks have been processed. Figure 2.4 shows a high level block diagram of the structure of Whirlpool. Figure 2.5 shows a more detailed

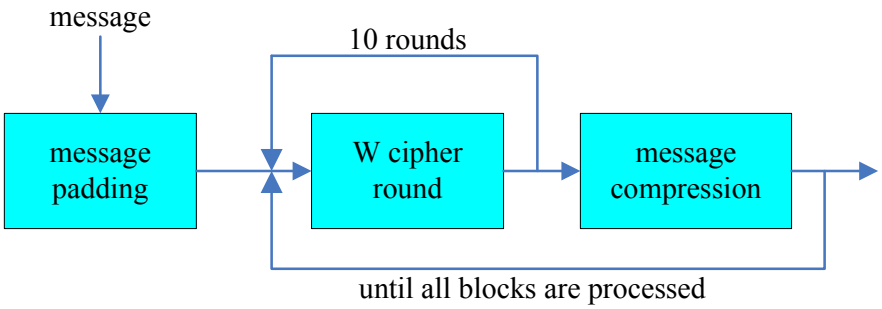

Figure 2.4: *High level block diagram of Whirlpool*

structure of Whirlpool. Each round consists of four distinct stages: the non-linear stage γ, the cyclic permutation stage π, the linear diffusion stage θ and the key addition stage

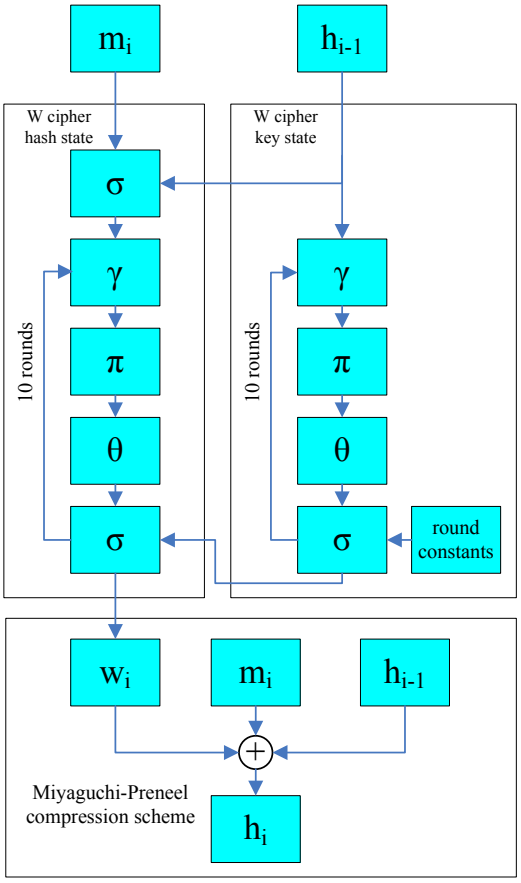

Figure 2.5: *Detailed block diagram of Whirlpool [20]*

σ. Initially the key addition stage σ is performed followed by all four stages performed sequentially. Each round also contains the key schedule which uses the four stages previously mentioned.

2.2.3 Non-linear stage γ

The non-linear stage γ, simply remaps individual bytes to a different value. Each byte to be remapped is first divided into two 4-bit nibbles. The more significant nibble is used to represent a row index, and the less significant nibble represents a column index. The row and column indices are then used as an address into a table. The table, called an S-box, is organized as a 16x16 matrix consisting of all 256 possible permutations of 8-bit values.

The current byte is then replaced with the byte at the given row and column index in the S-box. Table 2.8 defines the S-box. The labels along the left side of the table indicate the row indices and the labels along the top indicate the column indices.

	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
0_x	18_x	23_x	$C6_x$	$E8_x$	87_x	$B8_x$	01_x	$4F_x$	36_x	$A6_x$	$D2_x$	$F5_x$	79_x	$6F_x$	91_x	52_x
1_x	60_x	BC_x	$9B_x$	$8E_x$	$A3_x$	$0C_x$	$7B_x$	35_x	$1D_x$	$E0_x$	$D7_x$	$C2_x$	$2E_x$	$4B_x$	FE_x	57_x
2_x	15_x	77_x	37_x	$E5_x$	$9F_x$	$F0_x$	$4A_x$	DA_x	58_x	$C9_x$	29_x	$0A_x$	$B1_x$	$A0_x$	$6B_x$	85_x
3_x	BD_x	$5D_x$	10_x	$F4_x$	CB_x	$3E_x$	05_x	67_x	$E4_x$	27_x	41_x	$8B_x$	$A7_x$	$7D_x$	95_x	$D8_x$
4_x	FB_x	EE_x	$7C_x$	66_x	DD_x	17_x	47_x	$9E_x$	CA_x	$2D_x$	BF_x	07_x	AD_x	$5A_x$	83_x	33_x
5_x	63_x	02_x	AA_x	71_x	$C8_x$	19_x	49_x	$D9_x$	$F2_x$	$E3_x$	$5B_x$	88_x	$9A_x$	26_x	32_x	$B0_x$
6_x	$E9_x$	$0F_x$	$D5_x$	80_x	BE_x	CD_x	34_x	48_x	FF_x	$7A_x$	90_x	$5F_x$	20_x	68_x	$1A_x$	AE_x
7_x	$B4_x$	54_x	93_x	22_x	64_x	$F1_x$	73_x	12_x	40_x	08_x	$C3_x$	EC_x	DB_x	$A1_x$	$8D_x$	$3D_x$
8_x	97_x	00_x	CF_x	$2B_x$	76_x	82_x	$D6_x$	$1B_x$	$B5_x$	AF_x	$6A_x$	50_x	45_x	$F3_x$	30_x	EF_x
9_x	$3F_x$	55_x	$A2_x$	EA_x	65_x	BA_x	$2F_x$	$C0_x$	DE_x	$1C_x$	FD_x	$4D_x$	92_x	75_x	06_x	$8A_x$
A_x	$B2_x$	$E6_x$	$0E_x$	$1F_x$	62_x	$D4_x$	$A8_x$	96_x	$F9_x$	$C5_x$	25_x	59_x	84_x	72_x	39_x	$4C_x$
B_x	$5E_x$	78_x	38_x	$8C_x$	$D1_x$	$A5_x$	$E2_x$	61_x	$B3_x$	21_x	$9C_x$	$1E_x$	43_x	$C7_x$	FC_x	04_x
C_x	51_x	99_x	$6D_x$	$0D_x$	FA_x	DF_x	$7E_x$	24_x	$3B_x$	AB_x	CE_x	11_x	$8F_x$	$4E_x$	$B7_x$	EB_x
D_x	$3C_x$	81_x	94_x	$F7_x$	$B9_x$	13_x	$2C_x$	$D3_x$	$E7_x$	$6E_x$	$C4_x$	03_x	56_x	44_x	$7F_x$	$A9_x$
E_x	$2A_x$	BB_x	$C1_x$	53_x	DC_x	$0B_x$	$9D_x$	$6C_x$	31_x	74_x	$F6_x$	46_x	AC_x	89_x	14_x	$E1_x$
F_x	16_x	$3A_x$	69_x	09_x	70_x	$B6_x$	$D0_x$	ED_x	CC_x	42_x	98_x	$A4_x$	28_x	$5C_x$	$F8_x$	86_x

Table 2.8: *Whirlpool 16x16 S-box*

The S-box shown in Table 2.8 can be generated with the use of 4x4 mini S-boxes. Figure 2.6 shows the definition of the 16x16 S-box using such components. Each of the 4x4 S-boxes takes a 4-bit input and produce a 4-bit output. The mappings for the E, E^{-1} and R mini S-boxes are shown in Table 2.9. The column heading represents the input into the respective mini S-box. The value below a column heading represents the output of the specific mini S-box.

u	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
$E(u)$	1_x	B_x	9_x	C_x	D_x	6_x	F_x	3_x	E_x	8_x	7_x	4_x	A_x	2_x	5_x	0_x
$E^{-1}(u)$	F_x	0_x	D_x	7_x	B_x	E_x	5_x	A_x	9_x	2_x	C_x	1_x	3_x	4_x	8_x	6_x
$R(u)$	7_x	C_x	B_x	D_x	E_x	4_x	9_x	F_x	6_x	3_x	8_x	A_x	2_x	5_x	1_x	0_x

Table 2.9: *Mappings for E, E^{-1} and R mini S-boxes [20]*

The purpose of this stage is to introduce non-linearity into the algorithm. This stage is designed to exhibit no correlation between the linear combinations of input bits and the linear combinations of output bits. Another property this stage is designed to have is that

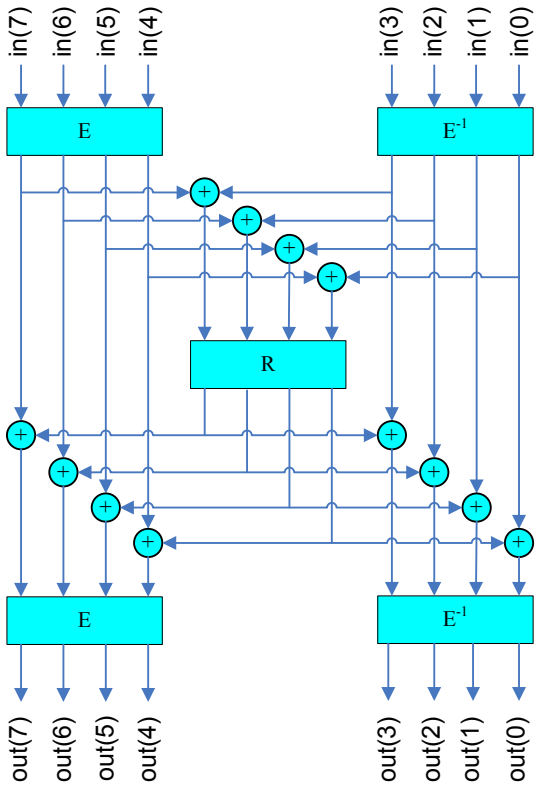

Figure 2.6: *S-box definition using mini S-boxes [20]*

differences between some set of input bits should not propagate into similar differences among the corresponding output bits. This property is also known as the avalanche effect, which refers to the idea that a small change in the input should cause a large change in the output.

2.2.4 Cyclic permutation stage π

In the cyclic permutation stage π, each matrix column j is cyclically shifted downward by j positions. Therefore the first column is left unchanged, the 2^{nd} column is shifted downward one position, the 3^{rd} column is shifted downward two positions, and so on. Since the input data is represented by an 8x8 matrix of bytes, column shifts move individual bytes from one row to another, which is a linear distance of a multiple of 8 bytes. Also note that this stage ensures that the 8 bytes of each row are spread out to 8 different rows.

2.2.5 Linear diffusion stage θ

The linear diffusion stage θ, achieves diffusion within each row individually. This stage consists of multiplication of the input data by a constant circulant matrix, C. A circulant matrix is a matrix where each row vector is rotated one element to the right relative to the preceding row vector. The matrix used in this stage is shown in Table 2.10.

$$C = \begin{bmatrix} 1 & 1 & 4 & 1 & 8 & 5 & 2 & 9 \\ 9 & 1 & 1 & 4 & 1 & 8 & 5 & 2 \\ 2 & 9 & 1 & 1 & 4 & 1 & 8 & 5 \\ 5 & 2 & 9 & 1 & 1 & 4 & 1 & 8 \\ 8 & 5 & 2 & 9 & 1 & 1 & 4 & 1 \\ 1 & 8 & 5 & 2 & 9 & 1 & 1 & 4 \\ 4 & 1 & 8 & 5 & 2 & 9 & 1 & 1 \\ 1 & 4 & 1 & 8 & 5 & 2 & 9 & 1 \end{bmatrix}$$

Table 2.10: *Circulant matrix used in the linear diffusion stage θ [20]*

In addition to being a circulant matrix, C is also designed to be a MDS matrix. Letting A represent the current Whirlpool hash state as an 8x8 matrix, the output from this stage will be B = A $*$ C , where $*$ denotes matrix multiplication. In the Whirlpool hashing algorithm, the matrix multiplication is performed in the Galois field, GF(2^8), modulo the reduction polynomial $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. Matrix multiplication consists of element-wise multiplication and addition. In GF(2^8) addition is simply the $\oplus$ operation. Multiplication in GF(2^8) is a more complicated process.

As an example, Equation 2.1 shows the computations required to determine the first entry, $b_{0,0}$, of the B matrix. $\otimes$ denotes multiplication in GF(2^8) and $\oplus$ denotes addition in GF(2^8). All remaining entries of the B matrix are computed in a similar manner.

$$b_{0,0} = a_{0,0} \oplus (9 \otimes a_{0,1}) \oplus (2 \otimes a_{0,2}) \oplus (5 \otimes a_{0,3}) \oplus (8 \otimes a_{0,4}) \oplus a_{0,5} \oplus (4 \otimes a_{0,6}) \oplus a_{0,7} \quad (2.1)$$

2.2.6 Key addition stage σ

The key addition stage σ, is the final stage in a single iteration of the W cipher. This stage simply performs the $\oplus$ operation between all bits of the hash state and all bits of the round key generated by the key schedule for that round. The output from this stage is used as the input to the next iteration.

Key schedule

The key schedule is used to generate the round key for the current round. In the Whirlpool hashing algorithm, not only is the hash value generated using the W cipher, the round key used in the key addition stage of each iteration is also generated using the same cipher.

An initial 512-bit key is used as the input to the key schedule algorithm. This key undergoes the non-linear stage γ, the permutation stage π and the linear diffusion stage θ in exactly the same manner as the hash state. The key addition stage σ for the key schedule differs slightly from the key addition stage of hash state calculation. The output of the linear diffusion stage θ undergoes the $\oplus$ operation with a round constant. The round constant changes depending on the iteration being executed.

The round constant starts out as an 8x8 matrix of bytes, consisting of all '0' bits except for the first row. The data in the first row of the round constant is replaced by data obtained from the 16x16 S-box shown in Table 2.8. The first 64 bytes, or 8 entries, of the S-box are copied into the first row of the round constant matrix to create the round constant for that round. To create the round constant matrices for the remaining nine rounds successive blocks of 64 bytes are copied from the 16x16 S-box, moving in a left-right and up-down fashion. The key generated in the current round is used as an input into the key schedule for the next round.

2.3 PHASH

PHASH [10] is a hash function designed to allow computations on data blocks to be performed in parallel. It consists of three stages: message padding, message compression and message reduction. Figure 2.7 shows a high level diagram of the PHASH hashing function.

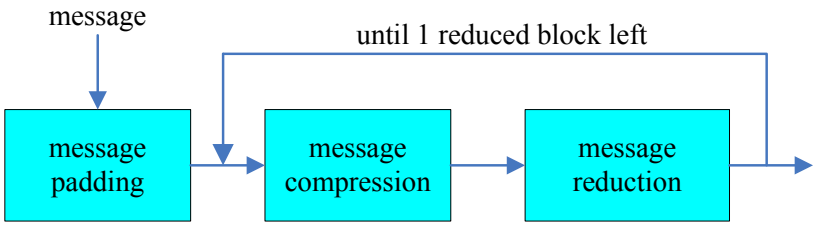

Figure 2.7: *High level block diagram of PHASH*

The message compression stage is where parallelism is used in order to increase throughput. PHASH relies on a block cipher for message compression. Any sufficiently large block cipher could be used for this purpose. For completeness, the W cipher, as presented in the Whirlpool hashing function, will be used throughout this work.

All three hashing functions discussed use a constant number of bits, N, to encode the message length during the message padding stage. In the case of SHA-512 and Whirlpool the choice of N does not affect the actual hash computations. However for PHASH the value of N does affect these computations. For completeness, the value of N used throughout this work will be 128. PHASH operates on messages less than 2^N bits in length.

Figure 2.8 shows a detailed block diagram of the PHASH hashing function.

2.3.1 Message padding

In order to ensure that the data to be hashed is aligned on the appropriate bit boundary a padding stage is performed on the data to be hashed. First a single '1' bit is appended to the end of the message, followed by a variable number of '0' bits. The number of '0' bits appended is such that the message length becomes a multiple of 768. Then the length of

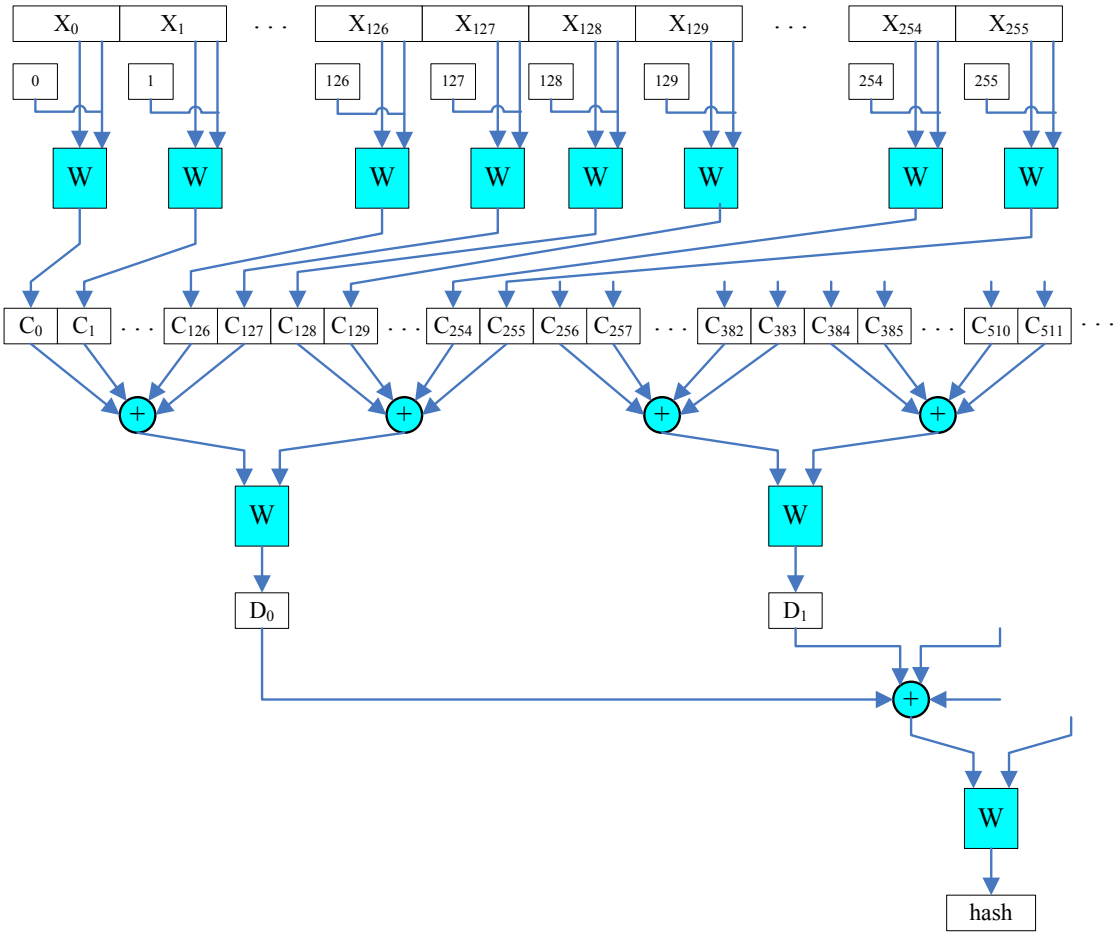

Figure 2.8: *Detailed block diagram of PHASH [10]*

the original message, before any padding has been done, is appended as an unsigned 128 bit integer. As a result, padding produces a message that is an exact multiple of 896 bits.

2.3.2 Message compression

The message compression stage takes each 896-bit block, X_i, and partitions it into two sub-blocks, Y_i and Z_i (not explicitly shown in Figure 2.8), where i represents the i^{th} block of the message to be hashed, after padding. Y_i consists of the 384 most significant bits of X_i, and Z_i consists of the remaining 512 bits. The Y_i block is concatenated with an unsigned 128 bit representation of i, producing a 512-bit block, V_i (not explicitly shown in Figure 2.8).

Recall that the W cipher takes a 512-bit plaintext input and a 512-bit initial key. For each Y_i block the V_i block is used as the plaintext to the W cipher, and the Z_i block is used as the key to the W cipher. The 512-bit compressed block C_i is the resulting ciphertext. The computation of the C_i's can be performed in parallel, since the computations are performed on independent blocks of data.

As previously mentioned the value of N has an effect on the hash computations. As N is increased the number of bits used for the i counter also increases and as a result every Y_i block will absorb less of the message to be hashed resulting in an increased hash computation time.

2.3.3 Message reduction

In the reduction phase the previously computed C_i blocks are reduced into D_i blocks. Computation of each D_i block involves using the computed C_i blocks. The first 128 C_i blocks undergo the $\oplus$ operation with each other. The next 128 C_i blocks undergo the same operation. If less than 256 C_i blocks are computed, the remaining C_i blocks consist of all '0' bits. As a result of the above process, 256 512-bit values are reduced to two 512-bit values. These two values are then fed into the W cipher as the key and plaintext, respectively, and a single 512-bit ciphertext, D_0 is produced as a result.

If more than 256 C_i blocks are computed, the next 256 C_i blocks are used in order to generate D_1 in the manner described above. The process continues until all D_i blocks have been generated. Then the same reduction process as previously described is applied to all D_i blocks until a single 512-bit value remains as a result of the reduction process. This final 512-bit value represents the hash of the original message.

Chapter 3

Supporting work

There are currently no papers published discussing the hardware implementation of the PHASH algorithm, however there have been many papers published discussing hardware implementations of the SHA-512, [1], [2], [5], [6], [9], [12], [13], [19] and Whirlpool algorithms [3], [11], [14], [18], [20]. The implementations were generally targeted for either high throughput or efficient resource utilization. Generally it is not possible to know, a priori, which design choice for a given component will be the best in achieving the specific design goal. Whenever possible several descriptions and implementations of specific components were presented. After implementation and execution of the algorithm with different components it was possible to comment on the quality of the given implementation as it pertained to achieving either a high throughput or low overall area.

3.1 SHA-512

This section includes information about currently existing hardware implementations of SHA-512. A summary and analysis of each implementation is included. The message expansion and message compression stages are described in detail. Most implementations acknowledge the existence of the padder stage, but do not include the details. Likewise, the hash update stage is mentioned, however its implementation is not always explicitly stated. One reason for omitting the details of the padder stage and the hash update stage is that it is assumed that these stages will be implemented in such a way as not to adversely affect

the overall performance of the algorithm.

3.1.1 Simple implementations

Some of the first published hardware implementations of SHA-512 are described in [9], [13] and [19].

In [19] the implementation of the $SHR(x)$ and $ROTR(x)$ operations used in both the message expansion and message compression stages of the algorithm is not discussed. ROM blocks are utilized to store the K_t constants required in the message compression stage. Several 64-bit adders along with simple combinational circuits were used to compute the values required by the algorithm. A five input, 64-bit, adder is used to compute the T_1 temporary value, and a two input, 64-bit, adder is used to compute the T_2 temporary value. The $Ch(x, y, z)$ and $Maj(x, y, z)$ functions are realized by simple combinational circuits.

A more detailed description of a SHA-512 implementation is provided in [13]. The design of the message expansion stage is shown in Figure 3.1.

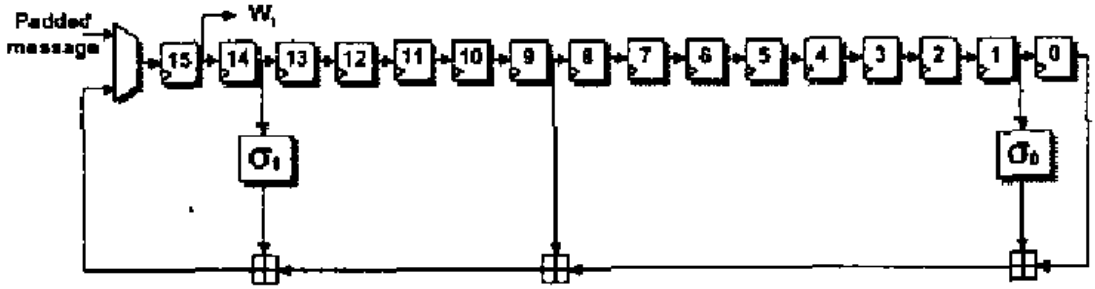

Figure 3.1: *Design of message expansion stage in [13]*

An advantage of this implementation is that it reuses registers. A naive implementation would require eighty 64-bit registers to store all of the data produced by the message expansion stage, however by sharing resources only 16 such registers and one multiplexer are required.

Each of the 16 registers is 64 bits wide. The multiplexer is used to choose when the first data block will be loaded into the registers. The output W_i from the leftmost register

is used as the input to the message compression stage. Once the first data block has been loaded into all the registers, the multiplexer feeds the outputs of the specified computation back into the leftmost register.

The design of the message compression stage in [13] is shown in Figure 3.2. Eight 64-

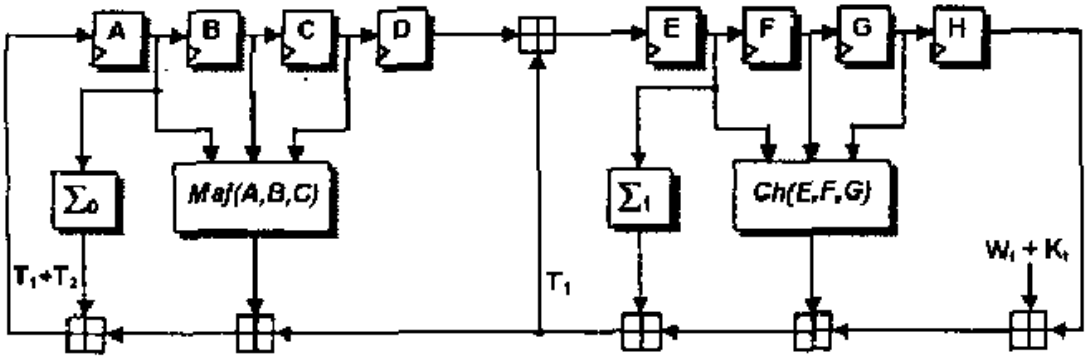

Figure 3.2: *Design of message compression stage in [13]*

bit registers represent the intermediate hash values which will be used in the hash update stage after 80 iterations of the compression stage have been performed. The implementation details of the $SHR(x)$ and $ROTR(x)$ operations have been omitted. In order to store the K_t constants used in this stage two dual-port BlockRAMs were used. BlockRAMs are on-chip memory components which can be instantiated and addressed like typical memory modules. Details about the adders used in this implementation have also been omitted.

In [9] an even more detailed implementation is presented. The design of the message expansion stage is shown in Figure 3.3. In order to reduce the total number of additions that need to be performed in the critical path the authors decided to compute the sum $K_t + W_t$ one cycle before it is needed. The R_{kw} register ensures that the sum is available during the correct iteration. The output of this register is used in the message compression stage. A combination of a CSA and CPA is used in order to minimize the amount of delay incurred by performing the addition of multiple 64-bit numbers. The delay would be much greater if only cascaded CPAs were used. The "LUT Shift A" and "LUT Shift B" blocks represent shift registers. They were designed using the "Shift Register Mode" of CLB slices available

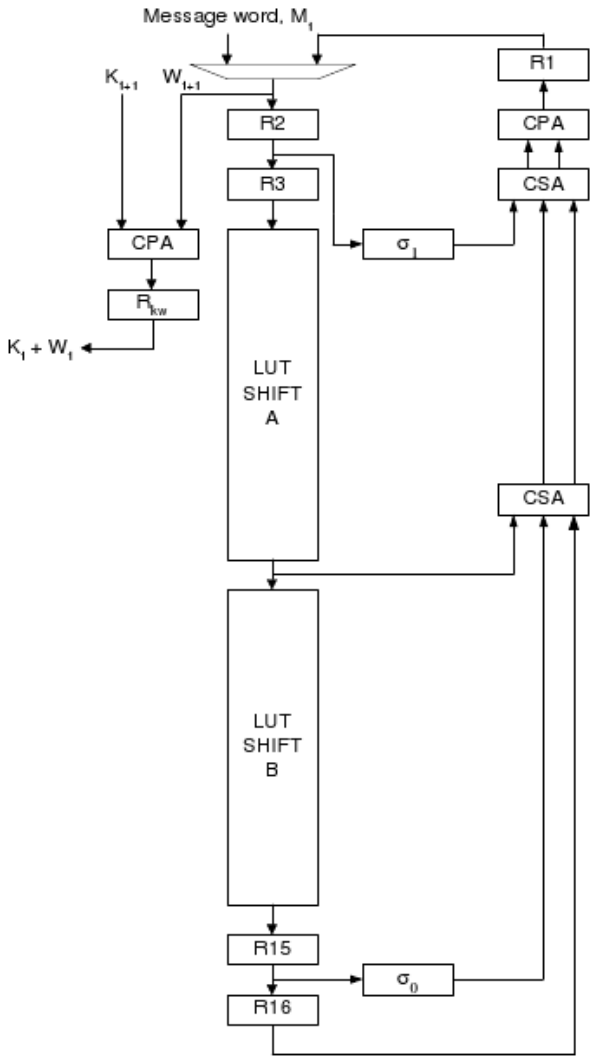

Figure 3.3: *Design of message expansion stage in [9]*

in the Virtex FPGA used for this implementation. They were designed this way mainly to reduce the overall area without necessarily affecting the operating speed.

The design of the message compression stage is shown in Figure 3.4. The multiplexers determine whether or not the current values of H_0^i through H_7^i will be used in the addition. The multiplexers select zeros to be added during all but the last round of the message compression stage. During the last round the current values of H_0^i through H_7^i will be used in the addition. The operation of these multiplexers performs the function of the hash update stage.

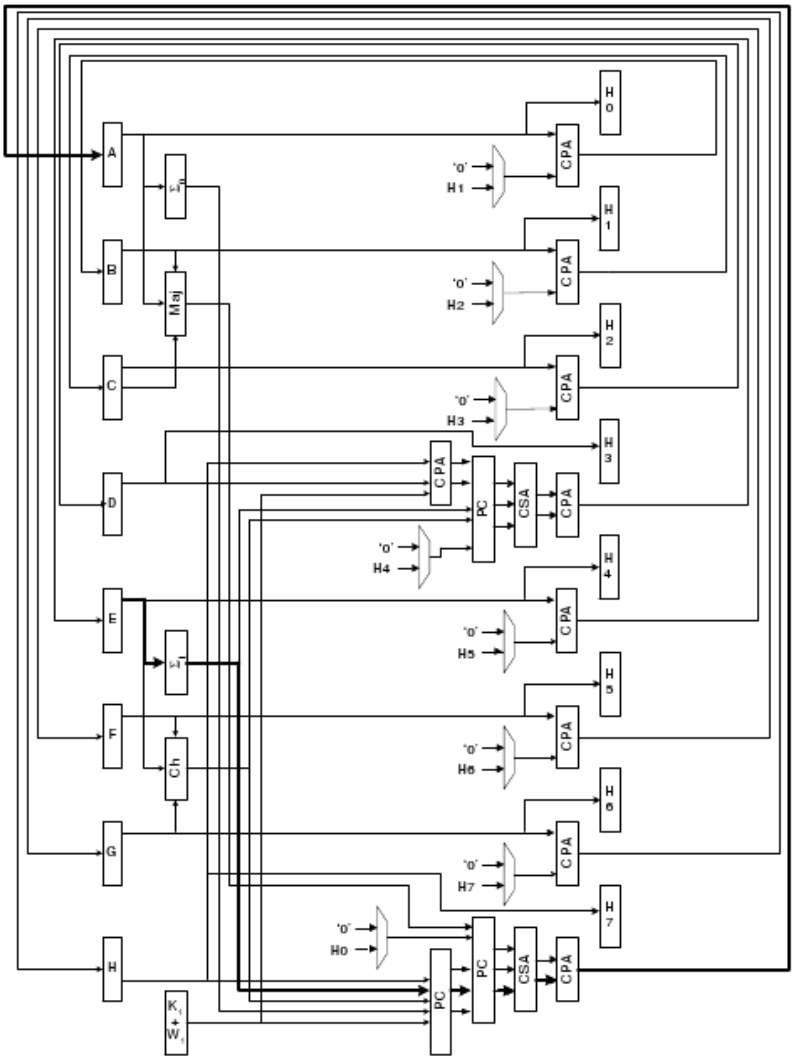

Figure 3.4: *Design of message compression stage in [9]*

The critical path in the message compression stage requires seven additions to be performed. In order to minimize the delay of this computation PCs, CSAs and CPAs are used. The parallel counter reduces the number of operands to be added from five to three. Once again, CSAs and CPAs are used in place of cascaded CPAs to minimize delay.

In [1] the basic SHA-512 design for the message expansion and message compression stages, as presented in [13] was used. CSA and CPA pairs were used to perform the addition of multiple 64-bit values. The main focus of this work however was to scale down the adders in order to minimize area. In addition to using the typical 64-bit adders 8-, 16-

and 32-bit adders were also used in order to perform the required additions.

3.1.2 More complex implementations

In [2] a partially unrolled SHA-512 implementation is presented. The message compression stage is unrolled two times. The rationale behind unrolling this stage is that six of the inputs, b through d and f though h feed the outputs directly. Only two of the inputs a and e require additional computation that introduces a delay into the algorithm. Unrolling allows two iterations to be executed at once. In order for this to be accomplished the message expansion stage is also modified to allow two W_t values to be obtained from it at the same time. Two iterations of the message compression stage are shown in Figure 3.5. The thick lines represent the critical delay path. By unrolling the message compression stage the operating frequency decreases by approximately a factor of two, thereby defeating the purpose of unrolling. In order to increase the operating frequency multiple operations need to occur in parallel. The current cycles output calculation will be performed in parallel with the next cycles immediate results based on the current cycles outputs. In order to achieve this the four nodes marked with dark circles in Figure 3.5 are selected as the immediate results, Pre_1 through Pre_4, respectively. The results of nodes Pre_1 and Pre_2 will be pre-computed during the previous cycle, concurrently with the previous cycles outputs. Nodes Pre_3 and Pre_4 will be pre-computed during the current cycle. Figure 3.6 shows the modified implementation with the Pre_1 through Pre_4 blocks in place. In order to obtain Pre_1 through Pre_4, variables c through h are needed. Only the e variable is not produced directly by the inputs. This means that in order to pre-compute Pre_1 and Pre_2, the next cycles e variable needs to be pre-computed. The resulting, and final, design of the message compression stage is shown in Figure 3.7

In [6] a quasi-pipelined implementation of SHA-512 is introduced. The message compression stage is pipelined, as it dictates the critical delay of the algorithm. The basic message expansion stage is altered so as to decrease the delay of the individual computations. The delay balancing technique is used to achieve a decrease in the delay of this stage.

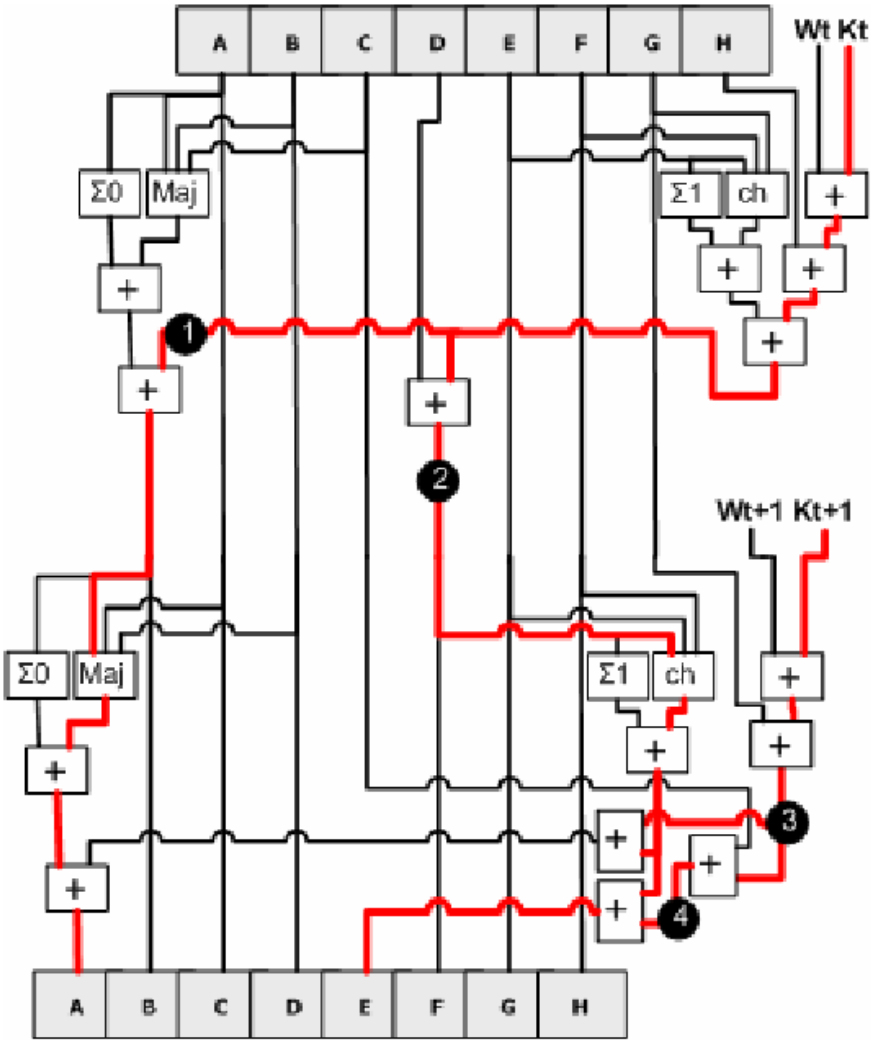

Figure 3.5: *Two iterations of the message compression stage in [2]*

To perform the required additions CSAs and CLAs are used. Note that all references to W_t in this section correspond to W_{ej} in the figures. (It is suspected to be a typographical error on the authors part.)

Figure 3.8 shows the design of the message expansion stage. Originally the CLA output was used as an input to the multiplexer. In order to reduce the delay of this stage it was relocated as shown in Figure 3.8. The first register is duplicated and a second multiplexer is added. During the first 16 cycles multiplexer $mux1$ is used for loading the block data into the register. During this time the input to $mux2$ is all zeros. After 16 cycles complete,

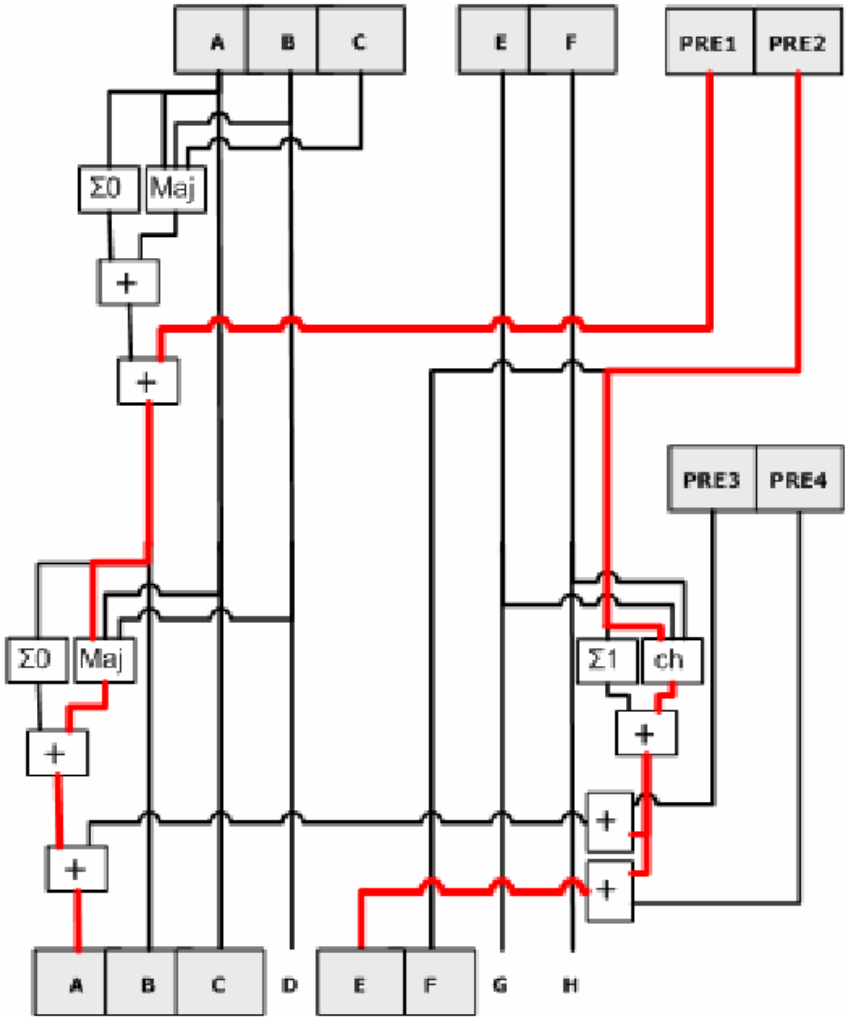

Figure 3.6: *Modified message compression stage with Pre_1 through Pre_4 blocks in place [2]*

both $mux1$ and $mux2$ select the outputs of CSA_2 to be used in the computations. In order to construct a pipelined implementation of the message compression stage the basic message compression stage implementation undergoes several changes. Figure 3.9 shows the first set of changes to the message compression stage. The definitions of all temporary variables used in Figure 3.9 are shown in Table 3.1. The output from the d register and the input to the adder generating the ϵ_1 output are initially left unconnected. They will be used once the pipelined implementation is finalized. The partitioning presented in Figure 3.9 lends itself to pipelining. Figure 3.10 shows the pipelined implementation of the message

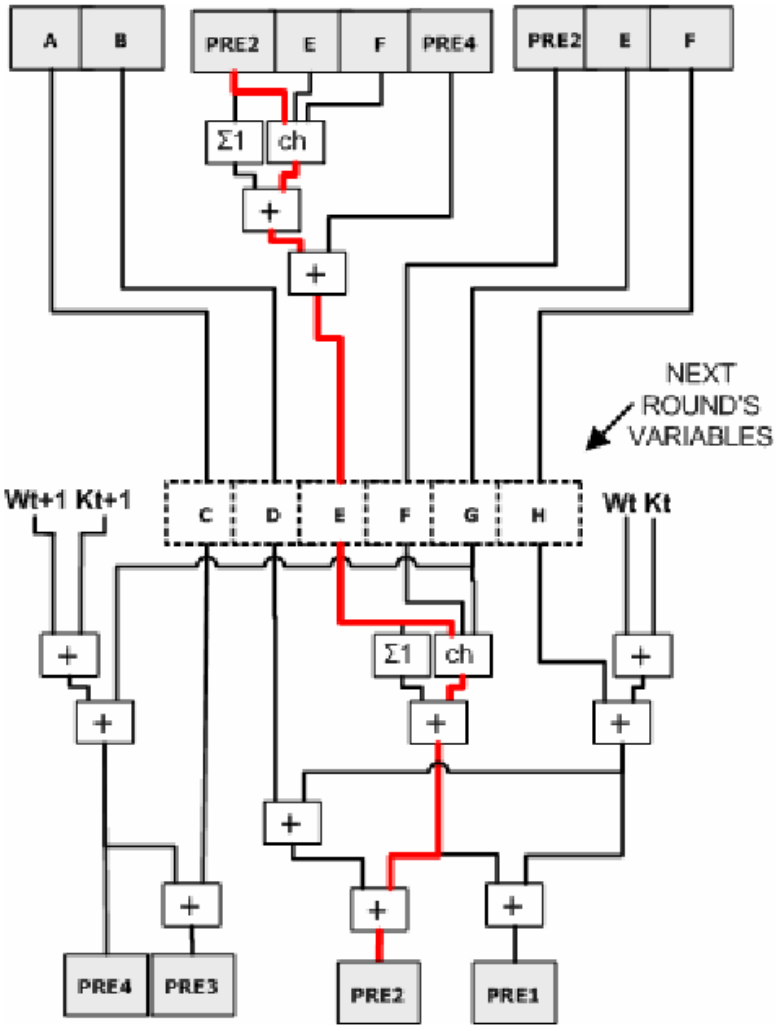

Figure 3.7: *Final design of the message compression stage in [2]*

compression stage. Registers M_1, M_2 and L define the three pipeline stages, τ_t,τ_{t-1} and τ_{t-2}. These registers store the results of ϵ_1, α_1 and α_2, respectively. The τ section is affected by variables W_t and K_t. The τ_{t-1} section is indirectly affected through τ by variables W_{t-1} and K_{t-1}. Likewise, the τ_2 section is indirectly affected through τ_t and τ_{t-1} by variables W_{t-2} and K_{t-2}. The multiplexer labeled $mux1$ is used to select the outputs from registers b through d. The multiplexer labeled $mux2$ is used to select the outputs from registers g and h. The two multiplexers select the appropriate inputs to the CSAs depending on the clock cycle being executed.

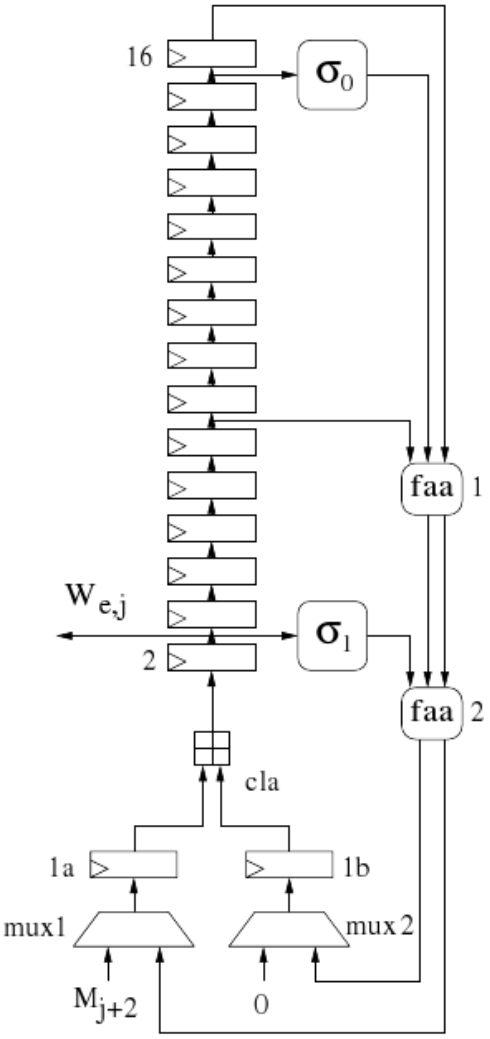

Figure 3.8: *Design of message expansion stage in [6]*

It is important to note that the pipelined implementation is only valid when the pipeline is full. Initially the registers a through h are loaded with the values shown in Table 2.3. During the 1st clock cycle the registers M_1 and M_2 are enabled, while register L is in a reset state. During the 2nd clock cycle e through h are clocked while a through d are inactive. The registers M_1 and M_2 are loaded with new values and register L is also loaded with its first value. In the 3rd clock cycle e through h get their first new values. The registers M_1, M_2 and L are still enabled and all three get their new values.

After the last W_t and K_t pair has been read, three more clock cycles are needed to

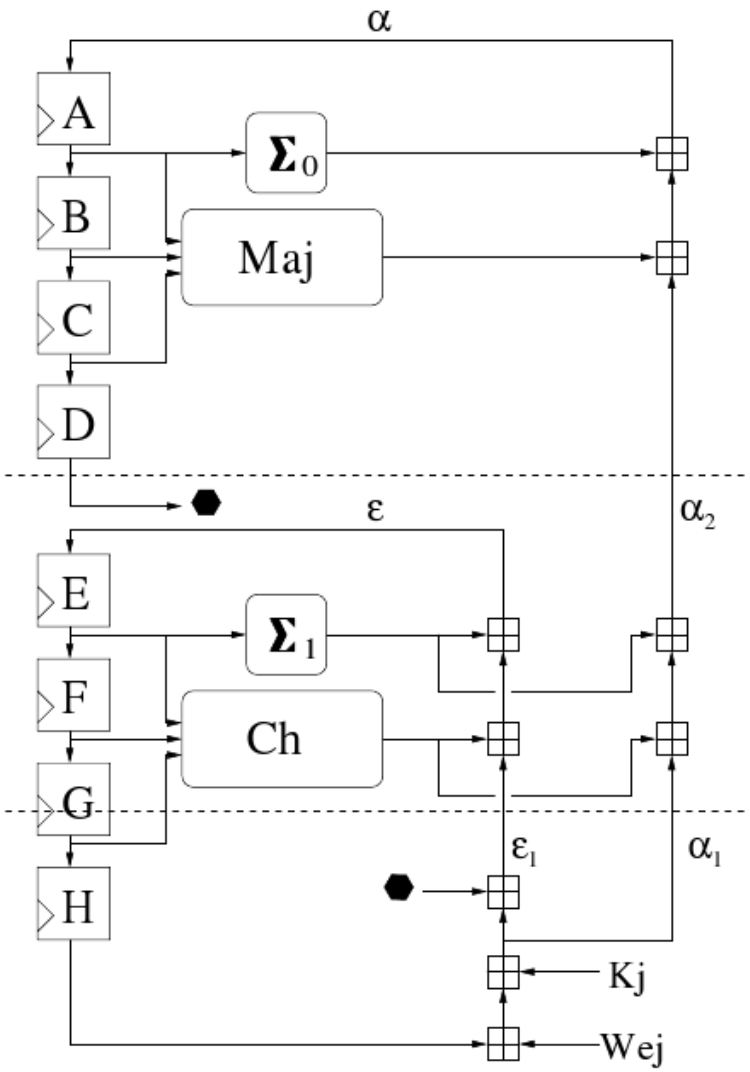

Figure 3.9: *First set of changes to the message compression stage in [6]*

finalize the message compression stage. The 1st cycle is used to transfer new values into registers M_1 and M_2. The 2nd cycle is used to generate the final values of e through h and the 3rd cycle is used to generate the values of a through d. It is important to note that in the 3rd cycle e through h must not be clocked.

In [12] yet another SHA-512 implementation is presented. The implementation draws on the collective strengths of most of the previously published work. It is quasi-pipelined and unrolled. The quasi-pipelined technique used is presented in [6]. The unrolling technique was introduced in [2]. The implementation uses CSAs and CPAs, as presented in

$$
\begin{aligned}
\alpha_1 &= W_t \oplus K_t \oplus h \\
\epsilon_1 &= \alpha_1 \oplus d \\
\epsilon &= \epsilon_1 \oplus Ch(e,f,g) \oplus \Sigma_1(e) \\
\alpha_2 &= \alpha_1 \oplus Ch(e,f,g) \oplus \Sigma_1(e) \\
\alpha &= \alpha_2 \oplus Maj(a,b,c) \oplus \Sigma_0(a)
\end{aligned}
$$

Table 3.1: *Definitions of temporary variables used in the message compression stage in [6]*

[9]. BlockRAMs are used to store the K_t constants, as presented in [13].

In [6] delay balancing is used in order to reduce the critical delay path of the message expansion stage, however such a technique is not suitable in conjunction with unrolling, so a new implementation of the message expansion stage was developed. Figure 3.11 shows the block diagram of the message expansion stage used in this implementation. Since this architecture is unrolled the values of W_t and W_{t+1} will be needed during the same clock cycle. The 16 64-bit registers which would normally be used in this stage are modified to be only 8 registers, but their width is doubled to 128 bits. The first 64-bit word of the new register represents values which will be used in the calculation of W_t, and the second 64-bit word will be used to calculate the value of W_{t+1}. The combinational logic of the message expander was duplicated so that W_t and W_{t+1} would be calculated in the same cycle. The multiplexer was also altered to select 128-bit values as opposed to 64-bit values. The W_t and W_{t+1} outputs are registered to ensure that they do not contribute to the critical delay path.

A diagram of the twice unrolled and pipelined message compression stage is shown in Figure 3.12. In this stage all of the logic needed to implement the functions used in a basic SHA-512 implementation is duplicated, however the multiplexer and the registers used to store the intermediate hash values are not. The registers are loaded differently in this architecture than in the basic one.

This is one of the only papers to describe the control circuitry used. The controller for

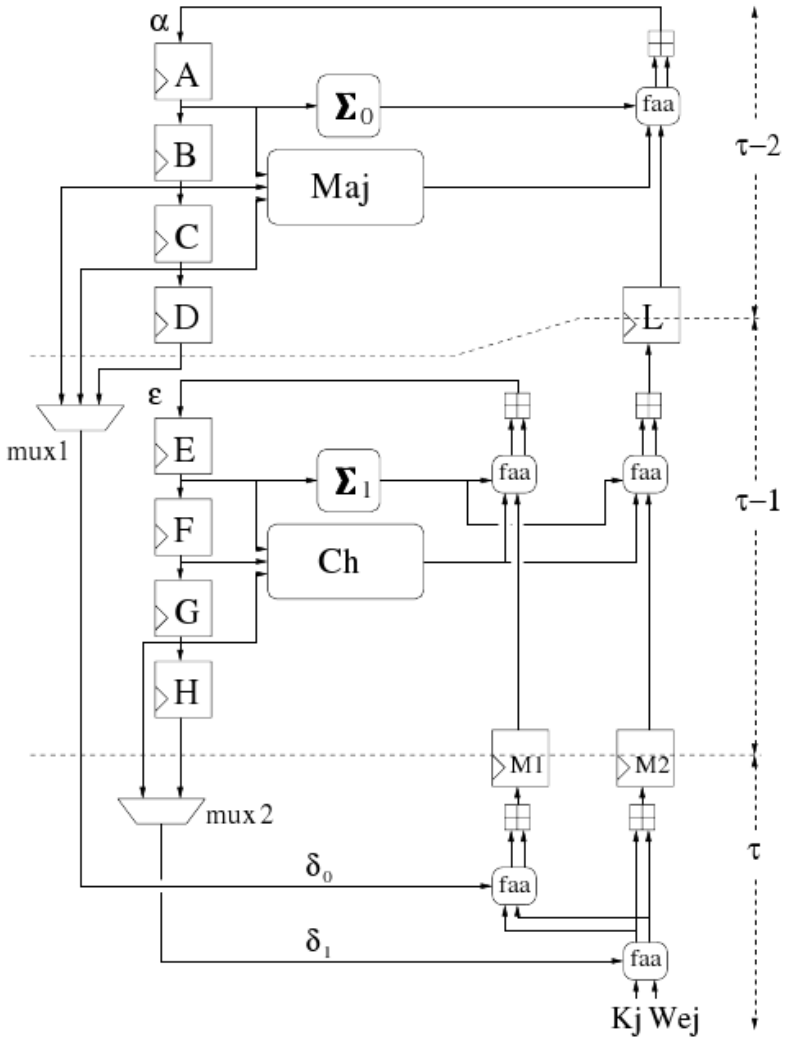

Figure 3.10: *Pipelined implementation of message compression stage in [6]*

this implementation was designed using a state machine. It is responsible for loading and clearing registers, controlling the multiplexer select lines as well as controlling the flow of data between the padder, the message expansion and message compression stages. It is also responsible for enabling and disabling the unrolled section of the message compression stage as required. All control signals from the state machine were registered to ensure they did not contribute to the critical delay path.

Since this design is pipelined, the unrolled section of the design cannot start executing until the pipeline has been filled. Similarly, the unrolled section cannot be used during the

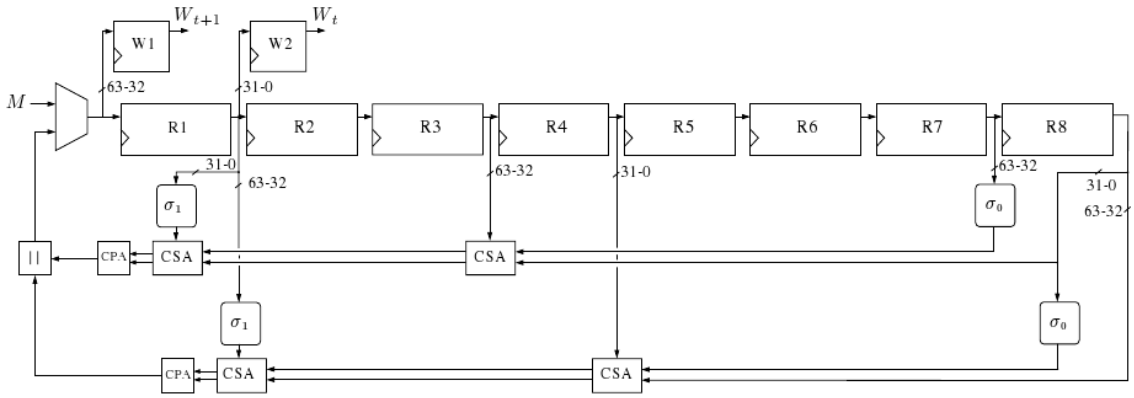

Figure 3.11: *Design of the message expansion stage in [12]*

final iteration of the message compression stage, since two halves of the final intermediate value appear in two consecutive clock cycles.

In [5] operation rescheduling was used. This technique also involves pre-computation in order to minimize the critical delay. The block diagram containing all of the components of the message compression stage is shown in Figure 3.13. The values of b through d and f though h are simply shifted around in order to generate new outputs. The computation of a and e needs a considerable amount of delay. Since the values of h, W_t and K_t are known before the message compression stage executes, their sum can be pre-computed. The value of h is simply the previous value of g.

Once 80 iterations of the message compression stage have completed the hash update has to be performed. Normally this would require eight 64-bit adders, however in [5] a scheme to accomplish the hash update using only two such adders was presented. Since most of the hash state variables are simply shifted around in order to generate new outputs relationships between the hash state variables were determined. Using such relationships six partial results of the hash update stage can be obtained by using only two adders. The remaining two values are calculated differently. These relationships are shown in Table 3.2. Using these relationships the partial results of the hash update stage can be expressed as shown in Table 3.3. As a result the necessary calculations can be performed using only two adders as shown in Table 3.4. Note that the values of H_0^i and H_4^i cannot be computed using

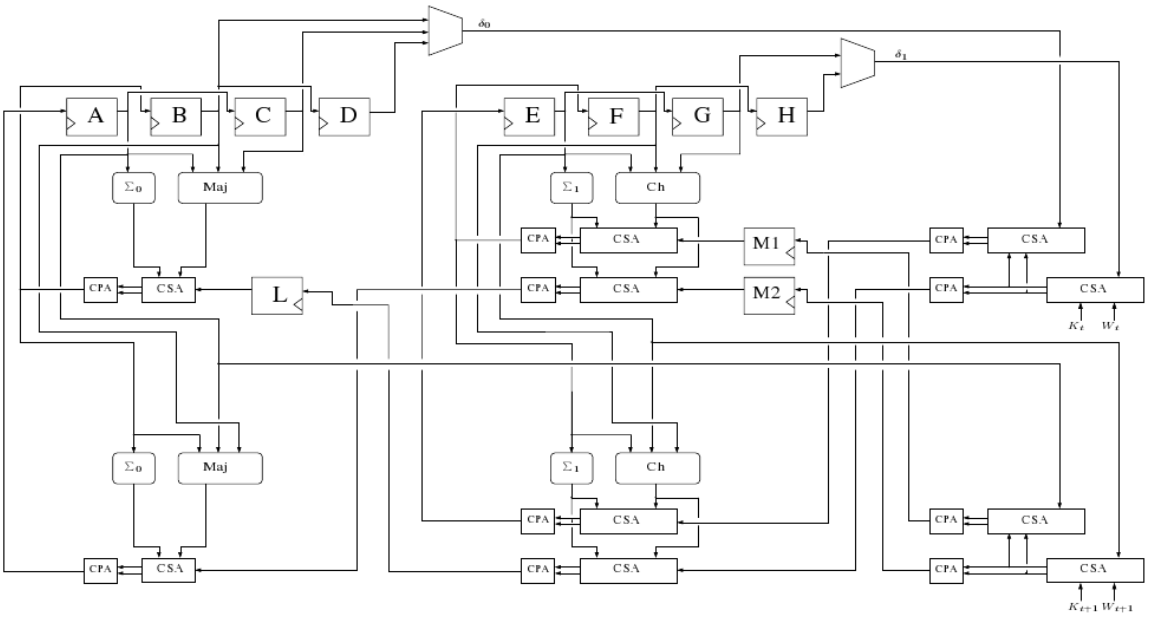

Figure 3.12: *Design of the message compression stage in [12]*

$$\begin{array}{ccccccc} h_t & = & g_{t-1} & = & f_{t-2} & = & e_{t-3} \\ d_t & = & c_{t-1} & = & b_{t-2} & = & a_{t-3} \end{array}$$

Table 3.2: *Relationships among hash state variables [5]*

this method. In order to compute H_0^i and H_4^i the values of a_t and e_t must be calculated first.

$$\begin{array}{ccc} H_1^i & = & H_1^{i-1} + a_{t-1} \\ H_2^i & = & H_2^{i-1} + a_{t-2} \\ H_3^i & = & H_3^{i-1} + a_{t-3} \\ H_5^i & = & H_5^{i-1} + e_{t-1} \\ H_6^i & = & H_6^{i-1} + e_{t-2} \\ H_7^i & = & H_7^{i-1} + e_{t-3} \end{array}$$

Table 3.3: *Partial results of the hash update stage [5]*

These values are calculated during the last iteration of the message compression scheme. Instead of using two full adders to compute H_0^i and H_4^i after the last iteration executes, H_0^i and H_4^i are computed during the calculation of a_t and e_t using CSAs and CPAs.

The six leftmost multiplexers in Figure 3.13 are used to select whether the working variables a through c and e through g or the results of the hash update stage will be used

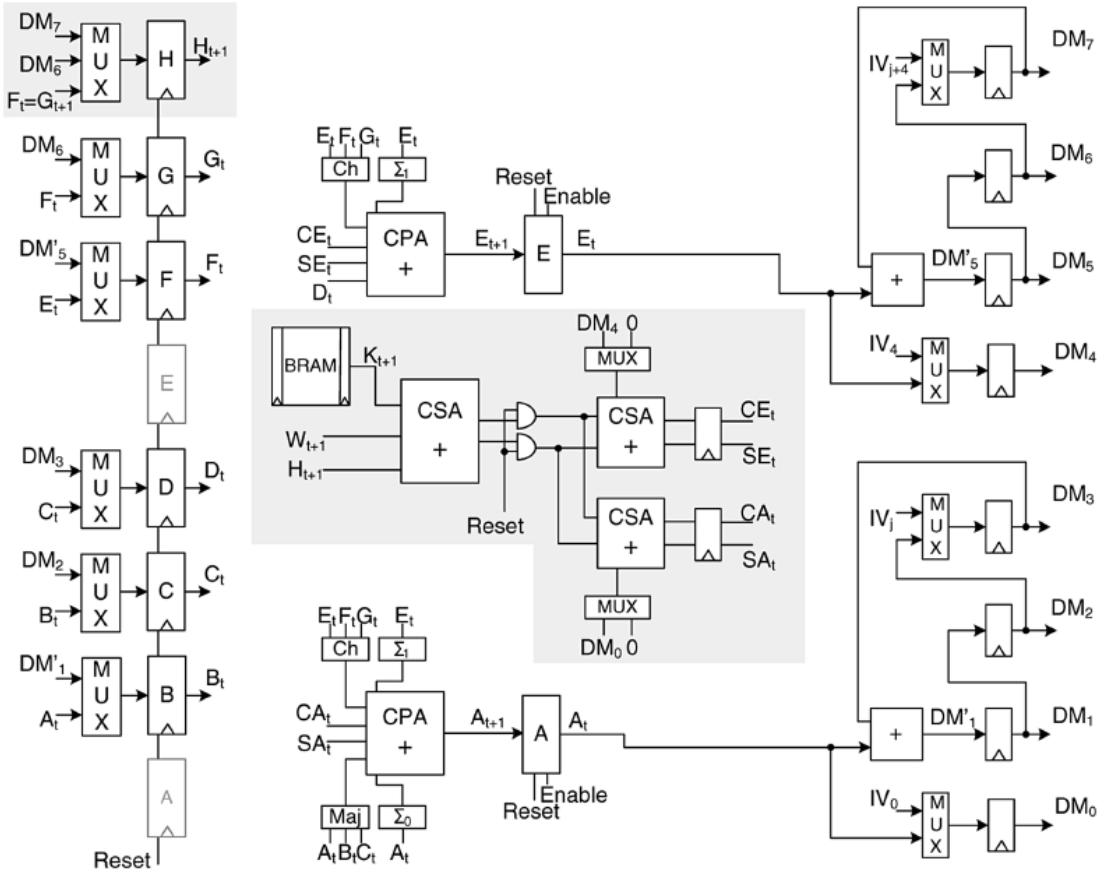

Figure 3.13: *Design of components in the message compression stage in [5]*

as inputs to the b through d and f through h registers. The four rightmost multiplexers are used during the initialization stage. If the initial hash state values are different from the ones provided in the SHA-512 specification the multiplexers are used to load the H_j^i registers with the desired values.

3.1.3 Efficiency of implementations

Table 3.5 summarizes the performance characteristics of all the SHA-512 implementations sorted by increasing throughput. The first three implementations provide a throughput of less than 1 Gbps and require a considerable number of CLB slices, as they were some of the initial implementations of SHA-512. Out of the three implementations providing the

$$H_x^i = a_{t-3+x} + H_x^{i-1} \quad 1 \leq x \leq 3$$

$$H_{x+4}^i = e_{t-3+x} + H_{x+4}^{i-1} \quad 1 \leq x \leq 3$$

Table 3.4: *Calculations necessary to perform partial hash update using only 2 adders [5]*

Implementation	Device	CLB Slices	Frequency (MHz)	Throughput (Mbps)	BlockRAM (bytes)
basic [19]	1	2237	75.00	480	N/A
basic [13]	2	2914	38.00	479	640
basic [9]	3	3441	55.56	670	640
basic [1]	4	4229	47.90	1226	N/A
basic [12]	5	2726	109.03	1329	640
unrolled [12]	5	4107	65.89	1466	1280
basic [5]	5	1666	121.00	1534	640
unrolled [2]	6	2582	60.50	1550	N/A
Device 1	Virtex V200PQ240				
Device 2	Virtex XCV1000E				
Device 3	Virtex XCV1000-6				
Device 4	Altera Stratix EP1S10F484C5				
Device 5	Virtex XC2V2000BF957				
Device 6	Virtex XCV1600EBG560				
N/A	information was not explicitly provided				

Table 3.5: *SHA-512 implementations summary*

highest throughput two of them were twice rolled. As a result they require more CLB slices for their implementations. The implementation provided in [5] proves to be the most efficient one in terms of both area and throughput. It requires only 1.09 CLB slices per 1 Mbps of throughput.

3.2 Whirlpool

This section includes information about currently existing hardware implementations of the Whirlpool hashing function. Also included is a summary and an analysis of these implementations.

3.2.1 Architectures

The core operation of the Whirlpool hash function consists of the W cipher as shown in Figure 3.14. This diagram also shows the padder, however none of the papers provide a detailed discussion of it. The implementations focus on the execution of the 10 rounds of the W block cipher given a single 512-bit block of data. Just as was the case with SHA-512, it is presumed that the implementation of the padder will not become the bottleneck in the execution of the algorithm.

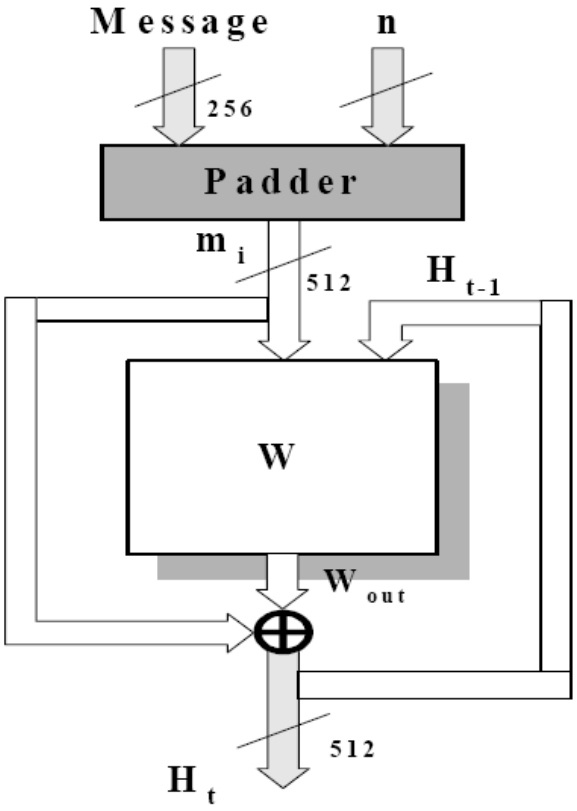

Figure 3.14: *Block diagram of the Whirlpool architecture [11]*

One thing to note is that the padder affects the overall execution time of a design depending on the amount of data it receives at one time. For example, if the padder receives 8 bits of data in one clock cycle it would take a total of 64 cycles to read in a single block, whereas if the padder receives 512 bits of data at once, only a single clock cycle is required

to read in the block. The padder in [11] receives 256 bits of data at one time. In [14] the amount of data read in by the padder at once was not mentioned.

Figure 3.15 shows a block diagram of a single round of the W cipher. This block diagram only includes the architecture of the hash state computation. This architecture is used in both [14] and [11]. Figure 3.16 shows the architecture of the W cipher with

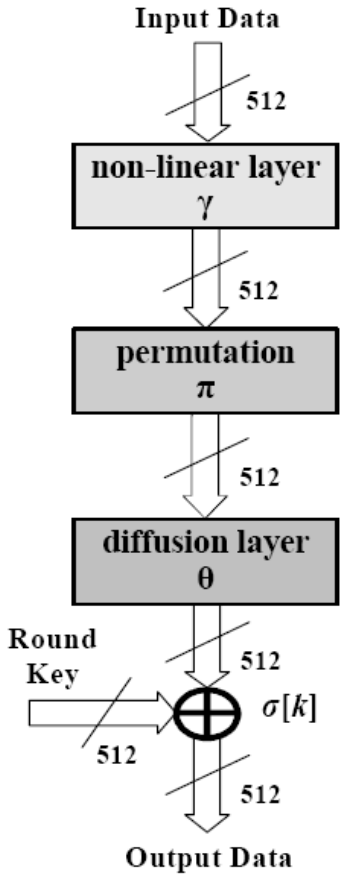

Figure 3.15: *Architecture of a single round of the W cipher [11]*

the key schedule in place. This architecture is used in both [14] and [11]. Since the key schedule uses the same W cipher used for the hash state computation, a 2nd architecture was proposed in [11] that reused the already existing W cipher from the hash state to implement the key schedule. Figure 3.17 shows the block diagram for this architecture. In [14] a twice unrolled architecture of the Whirlpool hashing function is introduced. Figure 3.18 shows the block diagram for such an architecture.

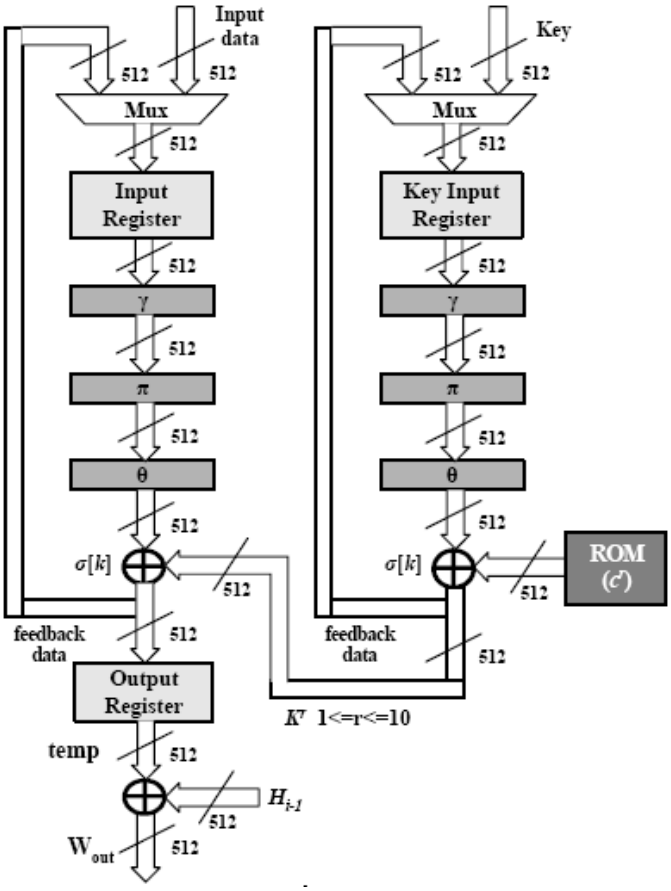

Figure 3.16: *Architecture of W cipher including the key schedule [11]*

Duplicated key schedule

In this architecture the computation of the hash and key states is performed in parallel. Since each phase contains the same number of stages, when the hash state computation is in the key addition stage σ, the key schedule will have produced the correct round key to be used. The advantage of such an architecture is that only 10 rounds are required to compute the hash value of a single block. The disadvantage is that it uses a considerably large area when implemented since the logic required for the key schedule is duplicated.

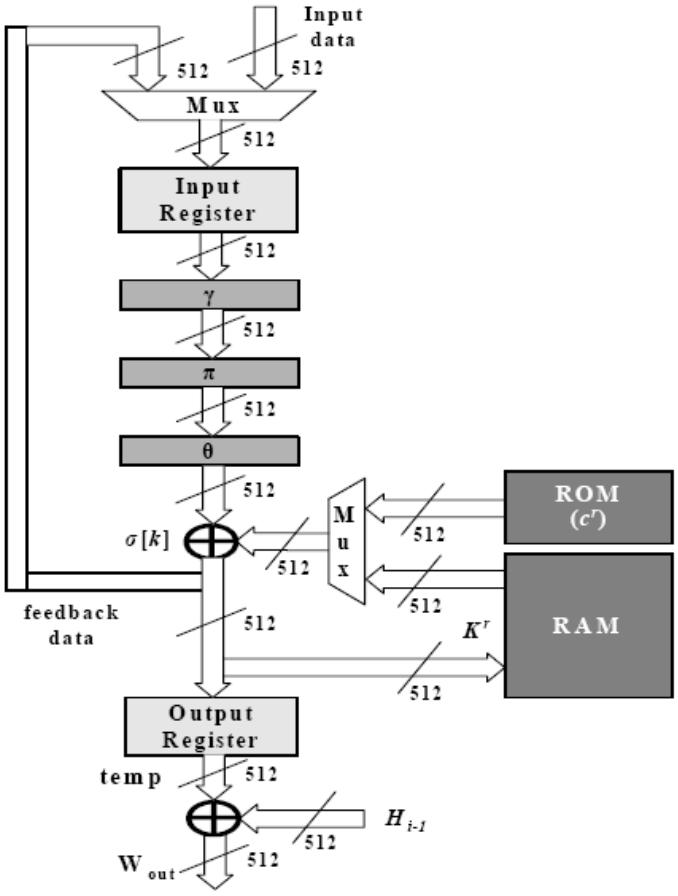

Figure 3.17: *Architecture of W cipher with incorporated key schedule [11]*

Incorporated key schedule

In this architecture the execution of the Whirlpool hash function is performed in two phases. The key schedule computations are performed first followed by the hash state computations. A ROM is used to store the predefined round constants used by the key schedule. The initial key is passed into the cipher and the resulting round key is stored in the RAM. Once all 10 round keys have been computed and stored in RAM, the hash state computation can begin. A multiplexer connected to both the RAM and ROM is used to select the appropriate data to be used. During the key schedule phase the round constants will be read from the ROM. During the hash state computation phase the precomputed round keys will be read from the RAM.

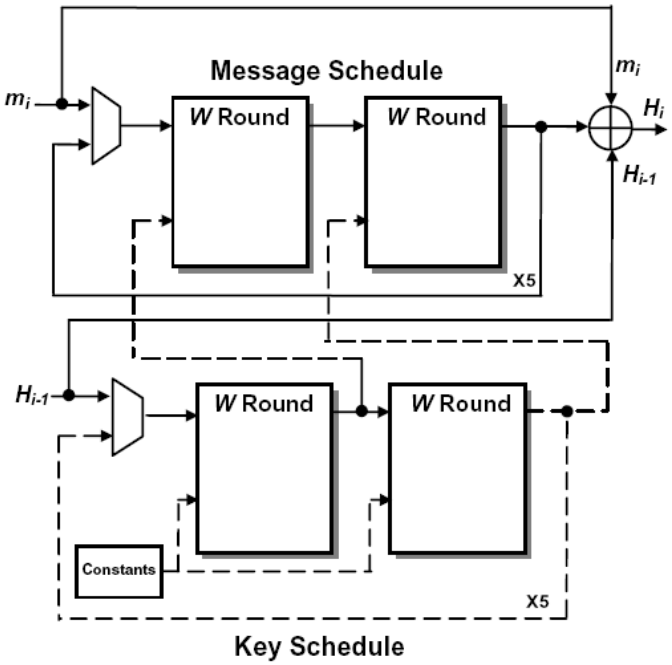

Figure 3.18: *Twice unrolled Whirlpool architecture [14]*

In this architecture the key schedule and the hash state has to be computed separately for every block. An advantage is that it requires much less overall area to implement since the W cipher can be reused for the key schedule. A disadvantage of such an architecture is that it takes twice as long to execute. This means that a total of 20 rounds, 10 rounds per phase, are required to compute the hash for a single block.

Twice unrolled

In this architecture the computation of two iterations can be performed in a single clock cycle. Such an implementation requires additional W cipher instances, thereby greatly increasing the overall area of the design. An advantage is that it only requires a total of 5 iterations in order to produce a hash value for a single block. It is important to note that halving the total number of iterations which need to be performed does not give a two-fold improvement in throughput. There is a fair amount of delay introduced with this architecture, since the output from the first iteration logic needs to propagate to the 2nd

iteration logic. There is additional delay added since the 2^{nd} iteration also needs to execute.

3.2.2 W cipher implementations

In this section the implementations of the four stages of the W cipher will be described in detail.

Non-linear stage γ

Three different ways to implement the non-linear stage γ, have been proposed in [14] and [11]. This stage consists of the implementation of the 16x16 S-box. One way to implement this stage is by using a full LUT to store all 256 8-bit values contained in the S-box. This stage was implemented using this approach in [14].

In order to achieve the highest throughput in this stage all of the lookups need to be performed in a single cycle. To meet this requirement all of the data needs to be accessible at once. In order for all the data contained in a single block to be substituted by the data in the S-box 64 instances of the S-box need to be instantiated; one instance per byte in a single block of data. The Virtex-4 LX100 FPGA used in [14] for the implementation of the Whirlpool hashing function contains BlockRAM. For this device the BlockRAM can be configured as a 256x8 bit dual-port RAM. By using the dual-port capability, two instances of the S-box can be mapped to a single BlockRAM cell. Therefore, in order to map all 64 instances of the S-box, 32 such BlockRAMs are needed. This results in a total memory usage of 16384 bytes.

A second method of implementing this stage involves the use of the three 4x4 mini S-boxes shown in Figure 2.6. This stage was implemented using this approach both in [14] and [11]. Each mini S-box was implemented using small LUTs separated by $\oplus$ operations. Each LUT consists of 16 8-bit values which can easily be mapped to the asynchronous distributed 16x1 bit ROM components found within a CLB slice on the Virtex-4 LX100 FPGA and also in most currently available FPGA devices. It is also possible to implement these mini S-boxes using the aforementioned BlockRAM components, however since each

BlockRAM is an 18 K-bit memory, these components would be greatly under-utilized.

The final proposed method to implement this stage was used in [11]. This method involves using boolean expressions to implement the small LUTs. Boolean expressions were developed that relate individual input bits to the corresponding output bits for each of the mini S-boxes. Table 3.6 shows a set of boolean expressions that were used to implement the mini S-boxes.

It is worth mentioning that these expressions are not unique and also may not utilize the least amount of logic when implemented in hardware. Table 3.6 contains expressions involving u_i, z_i and t_i. u_i represents the input bits into the mini S-box, z_i represents the output bits from the S-box and t_i denotes intermediate values. Using only $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT) and $\oplus$ (XOR) operations the given boolean expressions can be implemented using a total of 101 logic gates.

$z = E(u)$	$z = E^{-1}(u)$	$z = R(u)$
$t_0 = u_0 \wedge u_2$	$t_0 = \neg u_0$	$t_0 = \neg u_0$
$t_0 = t_0 \oplus u_1$	$t_1 = u_0 \vee u_1$	$t_1 = u_2 \wedge u_3$
$t_2 = \neg u_0$	$t_1 = t_1 \oplus u_3$	$t_2 = u_0 \oplus t_1$
$t_1 = u_3 \oplus t_2$	$t_2 = u_2 \wedge t_1$	$t_2 = t_2 \vee u_1$
$t_2 = t_0 \vee t_1$	$z_3 = t_0 \oplus t_2$	$t_3 = u_3 \vee t_0$
$z_0 = u_0 \oplus t_2$	$t_2 = u_0 \wedge u_2$	$z_2 = t_2 \oplus t_3$
$t_2 = u_2 \wedge t_0$	$t_3 = u_0 \vee u_3$	$t_2 = \neg u_2$
$t_1 = t_1 \oplus t_2$	$t_3 = t_3 \oplus t_2$	$t_2 = t_2 \oplus t_3$
$t_2 = u_3 \vee z_0$	$t_3 = t_3 \wedge u_1$	$t_3 = u_1 \vee t_2$
$z_1 = t_2 \oplus t_1$	$z_0 = t_0 \oplus t_3$	$z_3 = t_1 \oplus t_3$
$t_2 = u_2 \oplus t_1$	$t_2 = t_2 \oplus u_1$	$t_3 = t_3 \oplus t_0$
$t_1 = t_1 \oplus u_3$	$t_3 = u_2 \oplus t_0$	$t_0 = u_0 \vee z_3$
$t_0 = t_0 \oplus t_2$	$t_4 = z_3 \oplus t_2$	$t_1 = \neg u_1$
$t_1 = t_1 \vee t_0$	$t_1 = t_1 \wedge t_4$	$t_1 = t_1 \oplus u_3$
$z_2 = t_2 \oplus t_1$	$t_2 = t_2 \vee t_1$	$z_0 = t_0 \oplus t_1$
$t_1 = z_1 \wedge z_2$	$z_2 = t_3 \oplus t_1$	$t_3 = t_3 \vee z_0$
$t_1 = t_1 \vee z_0$	$t_0 = t_0 \vee u_3$	$z_1 = t_2 \oplus t_3$
$z_3 = t_0 \oplus t_1$	$z_1 = t_0 \oplus t_2$	

Table 3.6: *Boolean expressions used to implement the E, E^{-1} and R mini S-boxes [4]*

Linear diffusion stage θ

The linear diffusion stage θ, requires multiplication and addition to be performed in GF(2^8) modulo the reduction polynomial $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. Equation 2.1 shows the computations that need to be performed in order to compute a single coefficient , $b_{0,0}$, of the resulting matrix. Both [14] and [11] take advantage of the fact that multiplication by a power of 2 can be implemented as a simple left shift operation in hardware. By using this idea, several terms in Equation 2.1 were regrouped to take advantage of the aforementioned property. The resulting calculations for the $b_{0,0}$ entry of the B matrix are shown in Equation 3.1. This method is applied to all other terms in the B matrix in a similar manner. This simplification technique reduces the number of modular multiplications that need to be performed from 320 to 192.

$$b_{0,0} = a_{0,0} \oplus a_{0,1} \oplus a_{0,3} \oplus a_{0,5} \oplus a_{0,7} \oplus (2 \otimes a_{0,2}) \oplus [4 \otimes (a_{0,3} \oplus a_{0,6})] \oplus [8 \otimes (a_{0,1} \oplus a_{0,4})] \quad (3.1)$$

Whenever a modular reduction is necessary it is performed on the fly. This modular reduction is accomplished with an $\oplus$ operation of the value being multiplied and the hexadecimal representation of the reduction polynomial, $1D_x$.

In [11] look-up tables were used to implement the multiplication in GF(2^8). A table, $X[u] = x * u$, where u denotes the input into the table is used to obtain the result of multiplication by 2. The table $X^2[u] = X[u] * X[u]$, defines multiplication by 4 and table $X^3[u] = X[u] * X[u] * X[u]$ defines multiplication by 8.

The actual mappings used in the look-up table were not provided in [11]. Table 3.7 shows the equations used in order to compute all coefficients of the resulting matrix for this stage using the look up table approach. The subscript i ranges from 0 to 7.

Another way to perform the multiplications by powers of 2 is described in [18]. The value of the MSB is incorporated into the logic required to perform such multiplications. Three sets of boolean expressions were used to generate the results of the multiplications

$$b_{i,0} = a_{i,0} \oplus a_{i,1} \oplus a_{i,3} \oplus a_{i,5} \oplus a_{i,7} \oplus X[a_{i,2}] \oplus X^2[a_{i,3} \oplus a_{i,6}] \oplus X^3[a_{i,1} \oplus a_{i,4}]$$
$$b_{i,1} = a_{i,0} \oplus a_{i,1} \oplus a_{i,2} \oplus a_{i,4} \oplus a_{i,6} \oplus X[a_{i,3}] \oplus X^2[a_{i,4} \oplus a_{i,7}] \oplus X^3[a_{i,2} \oplus a_{i,5}]$$
$$b_{i,2} = a_{i,1} \oplus a_{i,2} \oplus a_{i,3} \oplus a_{i,5} \oplus a_{i,7} \oplus X[a_{i,4}] \oplus X^2[a_{i,5} \oplus a_{i,0}] \oplus X^3[a_{i,3} \oplus a_{i,6}]$$
$$b_{i,3} = a_{i,0} \oplus a_{i,2} \oplus a_{i,3} \oplus a_{i,4} \oplus a_{i,6} \oplus X[a_{i,5}] \oplus X^2[a_{i,6} \oplus a_{i,1}] \oplus X^3[a_{i,4} \oplus a_{i,7}]$$
$$b_{i,4} = a_{i,1} \oplus a_{i,3} \oplus a_{i,4} \oplus a_{i,5} \oplus a_{i,7} \oplus X[a_{i,6}] \oplus X^2[a_{i,7} \oplus a_{i,2}] \oplus X^3[a_{i,5} \oplus a_{i,0}]$$
$$b_{i,5} = a_{i,0} \oplus a_{i,2} \oplus a_{i,4} \oplus a_{i,5} \oplus a_{i,6} \oplus X[a_{i,7}] \oplus X^2[a_{i,0} \oplus a_{i,3}] \oplus X^3[a_{i,6} \oplus a_{i,1}]$$
$$b_{i,6} = a_{i,1} \oplus a_{i,3} \oplus a_{i,5} \oplus a_{i,6} \oplus a_{i,7} \oplus X[a_{i,0}] \oplus X^2[a_{i,1} \oplus a_{i,4}] \oplus X^3[a_{i,7} \oplus a_{i,2}]$$
$$b_{i,7} = a_{i,0} \oplus a_{i,2} \oplus a_{i,4} \oplus a_{i,6} \oplus a_{i,7} \oplus X[a_{i,1}] \oplus X^2[a_{i,2} \oplus a_{i,5}] \oplus X^3[a_{i,0} \oplus a_{i,3}]$$

Table 3.7: *Boolean expressions used to implement computation of the B matrix using a table lookup approach [11]*

by 02_x, 04_x and 08_x respectively. Table 3.8 contains the boolean expressions used to implement the multiplications. b is the result of the multiplication of the input number, a, by the given constant. This implementation was not used in either [14] or [11].

$b = 02_x \cdot a$	$b = 04_x \cdot a$	$b = 08_x \cdot a$
$b_0 = a_7$	$b_0 = a_6$	$b_0 = a_5$
$b_1 = a_0$	$b_1 = a_7$	$b_1 = a_6$
$b_2 = a_1 \oplus a_7$	$b_2 = a_0 \oplus a_6$	$b_2 = a_5 \oplus a_7$
$b_3 = a_2 \oplus a_7$	$b_3 = a_1 \oplus a_6 \oplus a_7$	$b_3 = a_0 \oplus a_5 \oplus a_6$
$b_4 = a_3 \oplus a_7$	$b_4 = a_2 \oplus a_6 \oplus a_7$	$b_4 = a_1 \oplus a_5 \oplus a_6 \oplus a_7$
$b_5 = a_4$	$b_5 = a_3 \oplus a_7$	$b_5 = a_2 \oplus a_6 \oplus a_7$
$b_6 = a_5$	$b_6 = a_4$	$b_6 = a_3 \oplus a_7$
$b_7 = a_6$	$b_7 = a_5$	$b_7 = a_4$

Table 3.8: *Byte multiplications by 02_x, 04_x and 08_x [18]*

Cyclical permutation stage π

In [11] the cyclical permutation stage π is implemented using combinational shifters. Each matrix column is cyclically shifted down by an amount equal to the column number, with indexing starting at 0. The details of the implementation of these combinational shifters

has not been included.

It is noted in [14] that the cyclic permutations simply involve the repositioning of bytes in the data matrix and as a result no actual logical or arithmetic operations are required. This stage is implemented simply by a hard-wiring of the data paths.

Key addition stage σ

The key addition stage σ, consists of simple $\oplus$ operations. The current hash state undergoes the $\oplus$ operation with the round key generated by the key schedule for the given round.

3.2.3 Efficiency of implementations

Several stages of the W cipher were implemented using different methods and a summary of the different implementation choices is outlined below.

In [14] the non-linear stage γ, was implemented using both a full look-up table placed in BlockRAMs, as well as by implementing the three mini S-boxes in the distributed RAM located in CLB slices. The twice unrolled architecture that used the mini S-box approach to implement the non-linear stage γ was also implemented. As a result three different Whirlpool implementations were obtained. The cyclic permutation stage π, was implemented with no additional logic; the data paths were hardwired to perform the reordering of the data. The linear diffusion stage θ, was implemented by shifting the input byte around and the modular reduction step was performed, if necessary, on the fly. The key addition stage σ, consisted of only $\oplus$ operations.

In [11] Whirlpool was implemented using the architecture that duplicates the key schedule, as well as the architecture which integrates the key schedule into the design. The non-linear stage γ, was implemented using both the mini S-box approach and the boolean expression approach. As a result four different Whirlpool implementations were obtained. The permutation stage π, was implemented using combinational shifters. The linear diffusion stage θ, was implemented by using a look-up table containing the results of the multiplications of all possible inputs by the reduction polynomial in GF(2^8). The key addition

stage σ consisted of simple $\oplus$ operations.

Table 3.9 summarizes the performance characteristics of all the Whirlpool implementations sorted by increasing throughput.

Implementation	Device	CLB Slices	Frequency (MHz)	Throughput (Mbps)	BlockRAM (bytes)
BE, IK [11]	1	3815	75.00	1920	N/A
SL, IK [11]	1	3751	93.00	2380	640
BE, DK [11]	1	5713	72.00	3686	N/A
SL, DK [11]	1	5585	87.50	4480	N/A
SL, DK [14]	2	7507	91.37	4678	N/A
FS, DK [14]	2	4956	93.56	4790	17408
SL, unrolled [14]	2	13210	47.80	4896	N/A
Device 1	Xilinx V1000EFG1156-8				
Device 2	Xilinx XC4VLX100				
BE	boolean expressions used in non-linear stage γ				
DK	duplicated key schedule architecture				
FS	full LUT used in non-linear stage γ				
IK	incorporated key schedule architecture				
SL	small LUTs used to implement mini S-boxes in non-linear stage γ				
N/A	information was not explicitly provided				

Table 3.9: *Whirlpool implementations summary*

It is important to note that the slowest Whirlpool implementation provides a higher throughput than the fastest SHA-512 implementation. The majority of Whirlpool implementations achieve throughputs of over 4 Gbps. The three fastest implementations vary slightly in terms of throughput, however they use considerably different amounts of CLB slices. One of the implementations provided in [14] proves to be the most efficient one in terms of both area and throughput. It requires only 1.03 CLB slices per 1 Mbps of throughput.

Chapter 4

Design and modeling

This chapter includes detailed explanations of the VHDL models of SHA-512, Whirlpool and PHASH, targeted for simulation. Each implementation includes a behavioral and a structural model. The behavioral models are used in order to verify the correctness of the algorithms. Each structural model contains multiple components. These models are used in order to obtain the maximum throughput for the respective hashing function.

None of the implementations include a padder. It is assumed that the data being provided to these algorithms has already been pre-padded externally. A 64-bit wide data bus is used in all implementations. A reasonable limit of $2^{32} - 1$ has been imposed as the maximum number of blocks to be processed for all algorithms.

This work is based on a modified version of an early PHASH draft. This implementation was started before the final draft of PHASH was available.

Several reasonable restrictions were placed on the PHASH implementation in order to facilitate its development. The PHASH description specifies the maximum message to be less than 2^{128} bits in length. To compute the hash of a message of this length would require 15 levels of reduction. For this implementation a maximum of three levels of reduction are allowed, thereby limiting the maximum size of the message to 15032385407 bits or approximately 1792 MB. Such a limitation is reasonable, because if 4 levels of reduction were allowed the maximum size of the message would be 448 GB.

The PHASH description calls for N-bit counters to be used at each level of the reduction tree however in this implementation these counters are only used at the first level of

reduction. Successive reduction levels use the outputs of the previous reduction as inputs.

4.1 Behavioral models

All behavioral models in this work are implemented using the same approach. Each model contains a state machine and data path. The state machine is used to control the higher level aspects of the algorithm, such as data buffering as well as to provide appropriate output signals. The data path includes the implementation of the specific hashing function. The data path is only executed once the required amount of data has been read in and is ready to be processed. The structure of the behavioral models is shown in Figure 4.1. The $do\_hashing$ object represents the data path.

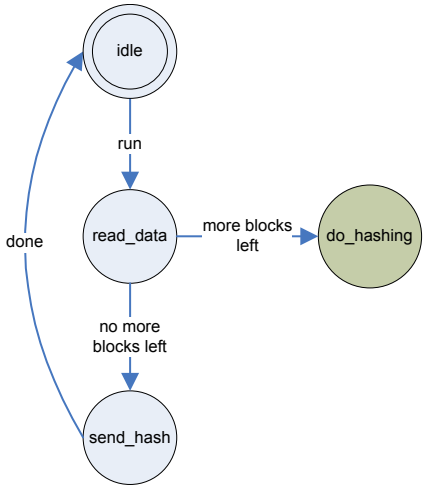

Figure 4.1: *Structure of SHA-512, Whirlpool and PHASH behavioral models*

All behavioral models begin in the $idle$ state. Once a run signal is asserted a transition to the $read\_data$ state occurs. Once a full block of data has been read the $do\_hashing$ process executes. Once all blocks have been processed a transition to the $send\_hash$ state occurs. In this state the final hash is output over 8 clock cycles.

The SHA-512 behavioral implementation contains a procedural implementation of the hashing algorithm using two for loops, logical operations and additions. The ROR and

SRL VHDL operators have been overloaded in order to facilitate the implementation of the algorithm.

The Whirlpool behavioral model contains a procedural implementation of the hashing algorithm. The non-linear stage γ is implemented using a look-up table. The cyclic permutation stage π is implemented using simple assignment statements. The GF multiplications required in the linear diffusion stage θ are also implemented using look-up tables. The key addition stage σ is implemented with a single $\oplus$ operation. The Miyaguchi-Preneel compression function is implemented with two $\oplus$ operations.

The PHASH behavioral model contains a procedural implementation of the hashing algorithm. A W cipher helper function is used in order to streamline the implementation. This function takes as input both a plaintext and an initial key, and returns a 512-bit hash.

4.2 Structural models

The structural models developed for each hashing function are described in detail in the following sections.

4.2.1 SHA-512

The implementation of the model in this work parallels the one described in [13]. It is not pipelined, and the critical delay path still consists of the addition of seven 64-bit values.

Two configurations of the model are included. The first configuration places the K_t constants, used in the message compression stage, into distributed RAM, and the second into a dual-port BlockRAM. The model consists of several distinct components as shown in Figure 4.2. Each component will be described in detail.

Message expansion

The expansion unit consists of a 1024-bit shift register, a 64-bit 2-to-1 multiplexer and three 64-bit adders. When this unit is enabled 64 bits of data is input into the shift register every

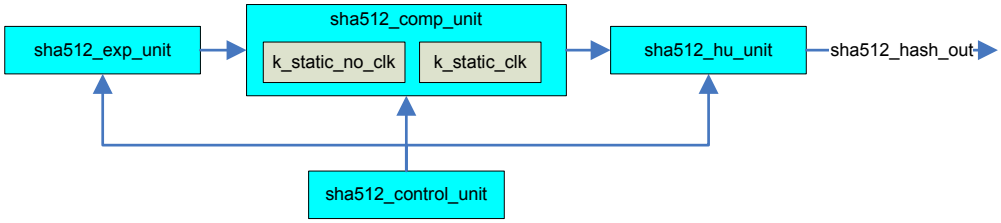

Figure 4.2: *Block diagram of the SHA-512 structural model*

clock cycle, t. The data input is defined by W_t as shown in Table 2.5. The multiplexer select line determines which set of data is to be input into the shift register. The value of W_t is an input into the compression unit.

Message compression

The compression unit consists of the registers a through h used to hold intermediate hash values, several adders as well as the component responsible for generating the 80 K_t constants. When this unit is reset, the values a through h get initialized as shown in Table 2.3. The values T_1, T_2 and a through h are updated every cycle, as shown in Table 2.6. The registers a through h can also be set externally. Their values are obtained from the output of the hash update unit. This is done so that these registers contain the newest intermediate hash. Depending on the chosen configuration the K_t generator will be implemented in either distributed RAM or as a single dual-port BlockRAM. The data in the BlockRAM is specified explicitly. The more significant 32 bits of each K_t constant are placed into the first 80 locations in the BlockRAM. The less significant bits are placed into the BlockRAM starting at location 128. By setting the address ports of the BlockRAM to $k\_address$ and $k\_address + 128$ the full 64-bit K_t constant can be obtained.

Hash update

The hash update unit consists of the registers H_j^i used to hold the intermediate hash values. When this unit is reset, the values H_0^0 through H_7^0 are initialized as shown in Table 2.3.

When this unit is enabled the inputs a through h are accumulated as shown in Table 2.7. This unit is also responsible for outputting the final hash. When all blocks have been processed the final hash value is output over 8 clock cycles.

Controller

The control unit is responsible for synchronizing the operation of all components in order to implement the SHA-512 hashing algorithm. It is implemented using a state machine. The state diagram for the control unit is shown in Figure 4.3.

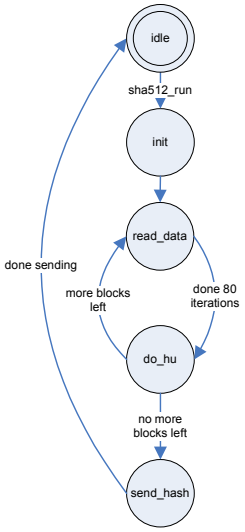

Figure 4.3: *State diagram of the SHA-512 control unit*

The control unit starts in the $idle$ state. Once the $sha512\_run$ signal is asserted a transition to the $init$ state occurs. At the same time data is ready to be read and the expansion unit is enabled. In the $init$ state the compression unit is enabled. On the next cycle a transition to the $read\_data$ state occurs.

This state is responsible for controlling the expansion, compression and hash update stages as well as keeping track of the number of blocks left to process. The multiplexer select line used in the expansion stage is assigned in this state. This state is also responsible for disabling the expansion stage, loading the compression stage and enabling the hash

update stage at the appropriate times. Once 80 iterations have occurred, a transition to the do_hu state occurs. During this transition the hash update occurs.

Once in this state, a transition back to $read\_data$ occurs if there are more blocks to process, otherwise a transition to $send\_hash$ occurs. The $send\_hash$ state is responsible for outputting the final hash over 8 clock cycles and asserting the $sha512\_done$ signal after the entire hash has been transmitted.

4.2.2 Whirlpool

The implementations for the model were obtained from descriptions provided in [11], [14], [18], [20]. Six configurations of the model are included as shown in Table 4.1.

Configuration	Non-linear stage γ	Mini S-boxes type	Linear diffusion stage θ
1	Mini S-boxes	Look-up table	Look-up table
2	Mini S-boxes	Look-up table	Boolean expressions
3	Mini S-boxes	Boolean expressions	Look-up table
4	Mini S-boxes	Boolean expressions	Boolean expressions
5	BlockRAM	N/A	Look-up table
6	BlockRAM	N/A	Boolean expressions

Table 4.1: *The six implemented configurations of the Whirlpool hashing function*

The model consists of several distinct components as shown in Figure 4.4. Each component will be described in detail.

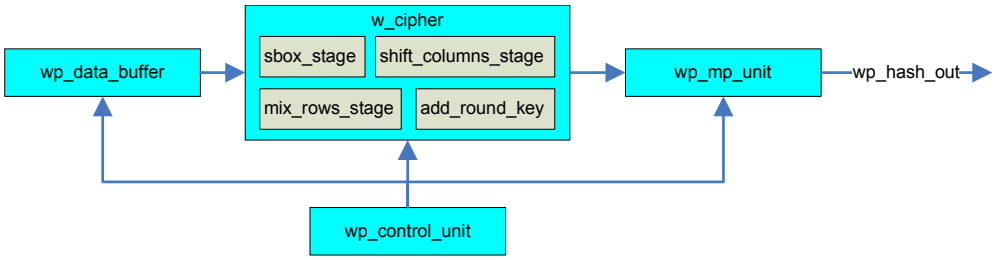

Figure 4.4: *Block diagram of the Whirlpool structural model*

Data buffer

The data buffer unit is a 512-bit shift register. It simply collects input data when it is enabled.

Non-linear stage γ

This unit provides two implementations of the S-box as shown in Table 2.8. The first implementation uses the mini S-boxes approach. The second implements the S-box using 64 single-port BlockRAMs. During synthesis these 64 BlockRAMs are converted into 32 dual-port BlockRAMs. Multiple $generate$ statements are used to accomplish this.

Mini S-box

Each of the three mini S-boxes, E, E^{-1} and R is implemented in two ways. The first is by using a look-up table, as shown in Table 2.9 placed in distributed RAM, and the second is using boolean expressions as shown in Table 3.6.

Galois field multipliers

The three units perform multiplication by 02_x, 04_x and 08_x in GF(2^8). Each unit is implemented in two ways. One implementation uses a look-up table and the other boolean expressions as shown in Table 3.8.

Cyclic permutation stage π

This unit performs the cyclic permutation on the input data. A $generate$ statement is used to accomplish this.

Linear diffusion stage θ

This unit provides an implementation of the GF matrix multiplication. Initially all input data is passed through each GF multiplication unit. Next the matrix multiplication is performed on a row by row basis. Eight $generate$ statements are used, one for each row of the resulting matrix. Within each $generate$ statement eight 8-bit values undergo the $\oplus$ operation.

Key addition stage σ

This unit performs the key addition step. In the hash state computation an $\oplus$ operation between the current hash state and current key state is performed. In the key state computation an $\oplus$ operation between the current key state and the appropriate round constant is performed.

W cipher

This unit provides the interconnections between the four stages required for the W cipher. Two implementations are included. The first implementation is used when the non-linear stage γ is implemented using mini S-boxes. The second implementation is used when the S-box is to be implemented within BlockRAMs. In this implementation a multiplexer is used in order to choose whether the newly loaded data or the output of the previous round is to be passed into the S-box.

Miyaguchi-Preneel compression

This unit is responsible for performing the Miyaguchi-Preneel compression after all 10 rounds of the W cipher have completed. The operation, as shown in Figure 2.3, is performed using two $\oplus$ operations. This unit is also responsible for outputting the final hash once all data blocks have been processed. The final hash is output over 8 clock cycles.

Controller

The control unit is responsible for synchronizing the operation of all components in order to implement the Whirlpool hashing algorithm. It is implemented using a state machine. The state diagram for the control unit is shown in Figure 4.5.

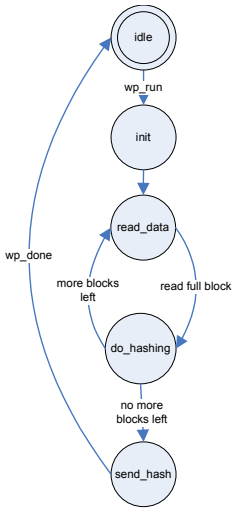

Figure 4.5: *State diagram of the Whirlpool control unit*

The control unit starts in the $idle$ state. Once the wp_run signal is asserted a transition to the $init$ state occurs. At the same time data is ready to be read and the data buffer is enabled. On the next cycle a transition to the $read\_data$ state occurs. This state is responsible for controlling the data buffer, loading and enabling the W cipher unit as well as keeping track of the number of blocks left to process.

Once an entire block of data has been read in a transition to the $do\_hashing$ state occurs. This state is responsible for controlling the W cipher and the Miyaguchi-Preneel compression unit. The W cipher is enabled for 10 rounds. The Miyaguchi-Preneel compression unit is enabled after 10 rounds have completed. This state also ensures that the correct round constant is used in the key addition stage σ. It also takes into account whether or not the non-linear stage γ is implemented using BlockRAMs. If it is not, no special measures are taken, however if BlockRAMs are used an enable signal for reading from them is generated.

After 10 rounds have completed and there are more blocks to process, a transition back to the $read\_data$ state occurs. If there are no more blocks to process, a transition to the $send\_hash$ state occurs. The $send\_hash$ state is responsible for outputting the final hash over 8 clock cycles and asserting the wp_done signal after the entire hash has been transmitted.

4.2.3 PHASH

The model consists of several distinct components as shown in Figure 4.6. Each component will be described in detail.

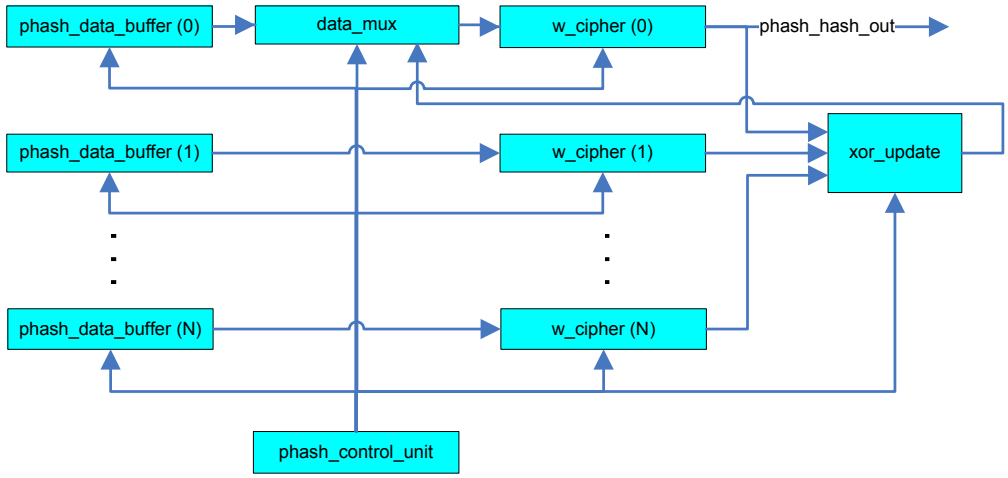

Figure 4.6: *Block diagram of the PHASH structural model*

PHASH can potentially utilize an arbitrary number of W cipher instances, however in order to simply the implementation, the number of instances was restricted to be a power of 2. Another simplification involved using a predetermined instance to perform reductions when the number of computed C_i blocks was a multiple of 256. This same instance was also responsible for producing the final hash after all data blocks have been processed.

Data buffer

The data buffer unit is an 896-bit shift register. It simply collects input data when it is enabled.

W cipher instances

This unit contains an implementation of the W cipher, as described in Whirlpool. This unit takes as inputs entire 512-bit plaintext and key blocks. It outputs the resulting 512-bit hash over a single cycle. In order to accomplish this the $read\_data$ and $send\_hash$ states were removed from the control unit as they were no longer needed. A multiplexer is added in order to ensure that the newly specified key is to be used at the beginning of the first round. This multiplexer is not needed in the original W cipher implementation because it is assumed that an initial key of all '0' bits is always used.

Data multiplexer

This unit consists of two 4-to-1 multiplexers. They are used to provide the appropriate plaintext and key to the W cipher units. The four inputs consist of the Z_i and V_i blocks and two $\oplus$ accumulators at each of the three reduction levels. These accumulators are named c_left, c_right, d_left, d_right, e_left and e_right.

XOR update

This unit accumulates the hashes of the compressed blocks at each reduction level. Two 512-bit registers are used at each reduction level, one to accumulate the results of the first 128 blocks, and the other to accumulate the results of the remaining 128 blocks at that reduction level. This unit also provides data to the first W cipher instance through the data multiplexer.

Controller

The control unit is responsible for synchronizing the operation of all components in order to implement the PHASH hashing algorithm. It is implemented using a state machine. The state diagram for the control unit is shown in Figure 4.7.

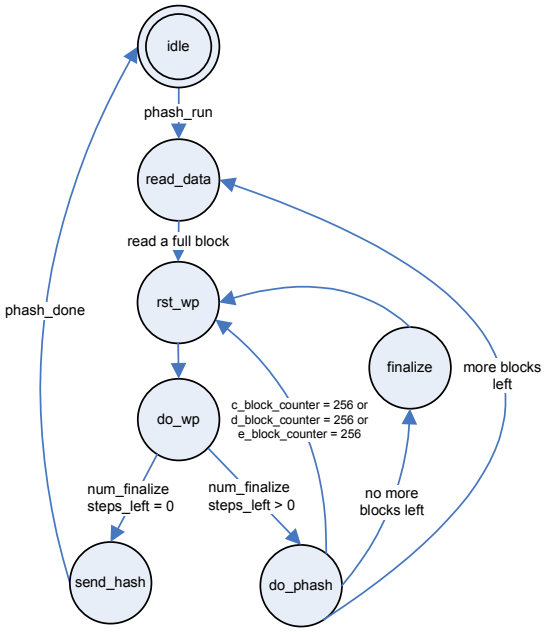

Figure 4.7: *State diagram of the PHASH control unit*

The control unit starts in the $idle$ state. Once the $phash\_run$ signal is asserted a transition to the $read\_data$ state occurs. At the same time data is ready to be read and the data buffer is enabled. Once an entire block of data has been read a transition to the rst_wp state occurs. At this time the W cipher instances are ready to be reset and the data multiplexer unit passes the Z_i and V_i blocks to the W cipher instances. This discussion assumes that the final hash is not yet being computed. A discussion of the final hash computation will follow.

During the rst_wp state certain W cipher instances are enabled, while others are disabled. If the total number of data blocks to be processed, $data\_num\_blocks$ is less than the number of instances, $num\_instances$ only $data\_num\_blocks$ instances are enabled, while

the others are disabled. Otherwise all W cipher instances are enabled until num_blocks_left becomes less than $num\_instances$. When this occurs only num_blocks_left instances are enabled, and the rest is disabled. After the appropriate instances are enabled a transition to the do_wp state occurs. The majority of the PHASH algorithm is controlled in this state. This state always waits for the wp_hash_ready signal from the W cipher instances. The control logic is broken down by the total number of reductions which need to be performed.

If only a single reduction level is required $c\_block\_counter$ is incremented every time a block is processed, and num_blocks_left is decremented accordingly. After a block is processed and $num\_finalize\_steps\_left$ is greater than 0, a transition to do_phash occurs, otherwise if the final hash has been computed a transition to $send\_hash$ occurs.

In the do_phash stage, if $c\_block\_counter$, $d\_block\_counter$ or $e\_block\_counter$ is less than 128 the resulting hashes will be accumulated into c_left, d_left or e_left respectively. Otherwise the results will be accumulated into c_right, d_right or e_right respectively. The appropriate counters are also incremented. If there are still more blocks to process and if the final hash is not being computed a transitions back to the $read\_data$ state occurs.

If two or more reduction levels are required the algorithm progresses as described above. However once $c\_block\_counter$, $d\_block\_counter$ or $e\_block\_counter$ equals 256, an additional hash needs to be computed using the appropriate registers from the XOR update unit as the key and plaintext. This is accomplished by a transition back to the rst_wp state. The computation is performed in the do_wp state. During this computation only the first W cipher instance is enabled. After the computation completes the appropriate set of registers is reset. On the next cycle a transition to the do_phash state occurs.

When there are no more blocks to process a transition from the do_phash state to the $finalize$ state occurs. After entering this state $num\_finalize\_steps\_left$ passes through the hashing states are performed. The value of $num\_finalize\_steps\_left$ depends on the highest reduction level attained. When finalizing the hash only the first W cipher instance is enabled. The initial key and plaintext inputs to this instance are c_left and c_right. If two reductions are to be done the next set of key and plaintext inputs are d_left and d_right.

Likewise, if three reductions are to be done the final set of key and plaintext inputs are e_left and e_right.

When the final hash is ready a transition to the $send\_hash$ occurs. The final hash is contained in the final hash output of the first W cipher instance. It is output over 8 clock cycles and upon completion the $phash\_done$ signal is asserted.

Chapter 5

Implementations for hardware

This chapter describes the development platform used in order to verify the functionality of the hashing function implementations. Changes to implementations targeted for simulation needed to be made in order to work correctly on this development board. These changes along with the communication interface between the PowerPC and the FPGA fabric will be discussed.

5.1 ML410 development platform

The available hardware platform to test the functionality of the designs is the Xilinx ML410 development board. A block diagram of this development board is shown in Figure 5.1. The ML410 is a complete development system containing a Virtex-4 FX60 FPGA as well as a multitude of additional peripherals. The most notable of these are the 64 MB DDR and 256 MB DDR-2 memories, 512 MB CF card, dual ethernet PHYs, a USB interface, SATA connectors, dual UARTs with RS-232 connectors as well as JTAG and trace debug ports.

The Virtex-4 FX60 FPGA is a hybrid FPGA, containing not only FPGA fabric, but also two PowerPC 405 cores. Several other notable features of this FPGA include 232 18-Kbit BlockRAMs, up to 20 DCMs, 25280 CLB slices, equivalent to 56880 logic cells, 128 XtremeDSP slices for accelerating matrix based operations as well as multi-gigabit RocketIO transceivers for implementing various communication protocols.

The features of the ML410 development board used in these implementations include

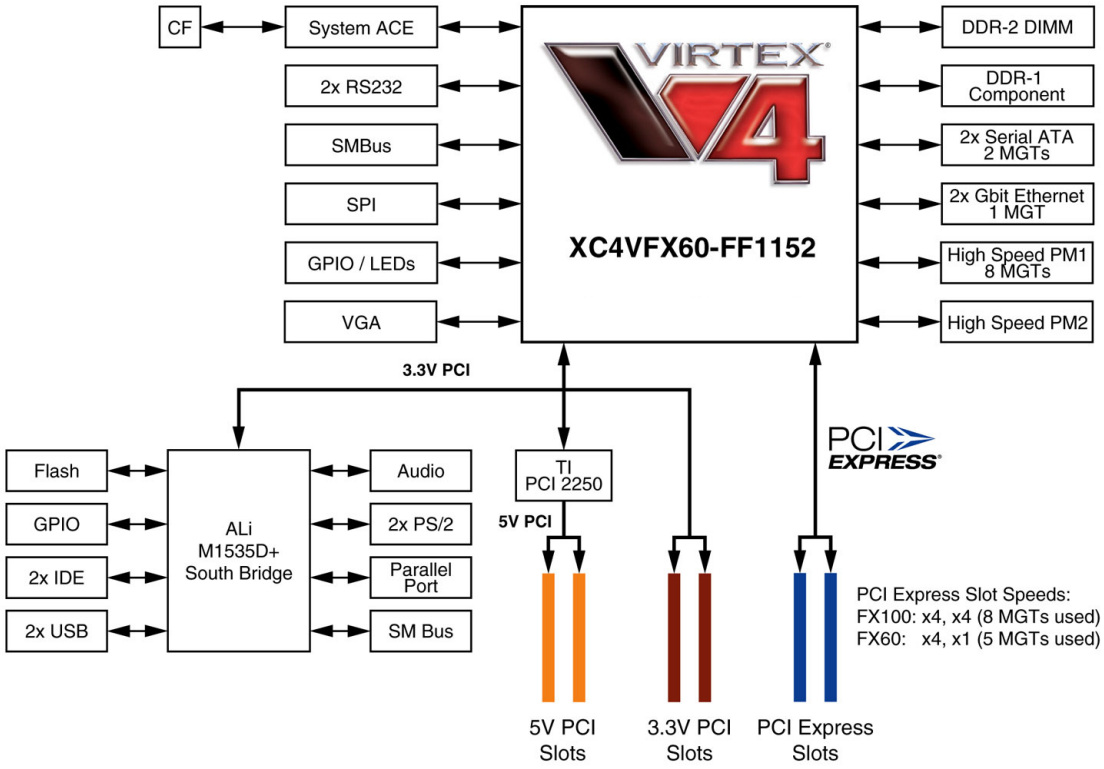

Figure 5.1: *Block diagram of the ML410 development board*

the 256 MB DDR-2 memory, 512 MB CF card and a single UART. The features of the Virtex-4 FX60 FPGA used include CLB slices, BlockRAMs and DCMs.

5.2 Base system

Xilinx EDK 9.2.02 software was used in order to design and implement the hardware systems required to test the functionality of the presented implementations. The EDK software provides IP cores for commonly used complex functions including digital signal processing, bus interfaces, processors and processor peripherals. Multiple cores can communicate with each other in order to establish a fully functional system. These cores can be added and configured manually within the EDK software, however doing so is a very tedious and

time consuming process. In order to reduce the development time, EDK provides a BSB wizard that allows the user to choose several common cores and configuration options for each core. The result is a fully functional base system. However, such a system is not very useful on its own. The EDK software allows for custom IP cores to be implemented and included in the base system. All hashing algorithms in this work will be implemented as a custom IP and included in the base system. A description of the base system common to all implementations is shown in Table 5.1.

Option	Value
FPGA Stepping	SCD1
Processor	PowerPC
Processor Clock Frequency	300 MHz
Bus Clock Frequency	100 Mhz
Processor Cache	Enabled
On-Chip Memory	16 KB
Debug Interface	FPGA JTAG
DDR-2 Memory	100 MHz with ODT
RS-232 UART	57600 baud
SysAce CF Interface	Enabled

Table 5.1: *Base system components and parameters common to all implementations*

The FPGA stepping is a way to signify the level of improvement over the initial release of the product. The higher the stepping level the more improvements have been made. Each base system includes a single PowerPC processor clocked at 300 MHz. The data bus is set to operate at 100 MHz. This bus is responsible for providing a communication channel between the PowerPC and all IP cores. The processor cache is enabled for improved performance. A total of 16 KB of on chip memory is allocated, allowing for code up to 16 KB in size to be stored and executed directly from this memory. A JTAG debug interface is included in order to enable optional debugging with the Xilinx XMD Debugger, or with a product such as Xilinx ChipScope. These debugging tools were not required for the implementation of these systems, so they will not be discussed further.

The 256 MB DDR-2 memory uses a 100 MHz memory clock, effectively providing an I/O bus clock speed of 200 MHz. ODT provides the necessary resistive termination on

the memory chip itself rather than externally on the motherboard. As a result the signal noise is reduced. With this feature disabled the DDR-2 memory self test routines failed. Consequently, none of the systems worked correctly with ODT disabled.

The RS-232 UART interface allows for data from the system to be output to a terminal. This data will be used to check whether or not the implementations are functioning correctly. The SysAce CF interface allows the PowerPC access to data contained on the provided 512 MB CF card. Data used to test the implementations will be placed on this CF card.

5.3 Custom IP wizard

Once the base system is complete, custom IP needs to be added implementing the hashing functions. The custom IP wizard is first used to create a template for the peripheral. It is later used to import the peripheral into the design. When creating the template several configuration options to increase the functionality of the peripheral are available. The three main features used for creating these peripherals include a write FIFO structure, software addressable registers as well as burst / cache-line support using a configurable data width. All implementations, excluding PHASH with 2 W cipher instances, use the same set of components and configuration options.

The write FIFO is used to buffer data received from the PowerPC. This is the data that will be passed to the respective hashing functions for processing. The data width of the FIFO is determined by the value of the data width selected as part of the burst / cache-line support feature. For all implementations a data width of 64 bits is used. The number of entries in the write FIFO is also selectable. For all implementations, excluding PHASH using 2 W cipher instances, a total of 2^{14} or 16384 number of entries are used. This allows a total of 131072 bytes to be stored in the write FIFO. The PHASH implementation using 2 W cipher instances contains 2 write FIFOs, both using a total of 2^{13} or 8192 number of entries. Using two write FIFOs allows both W cipher instances to begin processing

simultaneously. The individual FIFOs holds less data, however, since there are two of them the total amount of data that can be stored is still 131072 bytes. More details about this implementation will be provided in the next section.

The software addressable registers are 32-bit general purpose registers located in the peripheral that can be read from and written to externally from a software application. A total of 12 such registers are included in each peripheral.

Once the wizard completes, a stub is created for the given peripheral. The stub contains a basic implementation of reading and writing to the software addressable registers as well as a simple FIFO example. This code needs to be modified in order to include the desired functionality. In this case the desired functionality will be implementations of the three hashing functions.

Each of the hashing function implementations consist of several source files, and dealing with these files within EDK is a rather cumbersome process. In order to simplify the process of importing such peripherals, an NGC file will be created for each configuration using XST. XST is the tool used to synthesize HDL source code. The NGC file is essentially a netlist produced as a result of synthesis. This single file can now be used instead of all of the individual source files. Once the NGC file is obtained, the corresponding component needs to be instantiated inside of the peripheral stub code. At this point the write FIFO and the software addressable registers become important.

The wr_en signal of the write FIFO is managed internally. Simply by calling the appropriate function in the software application the specified data will be written to the FIFO. The rd_en signal is generated by the actual hashing function component being used. The way this signal is generated will be discussed in the next section. The rd_en signal is used to read data from the FIFO. This data will be processed by the respective hashing function.

Eight software addressable registers are used to store the final 512-bit hash so that it can be read by the software application after the $done$ signal is asserted. Registers are also used to provide the $start$ and $reset$ signals as well as to obtain the $done$ output when processing finishes. Another register is used to provide the hashing function with the number of data

blocks to be hashed.

Once all the necessary modifications to the peripheral stub code are made the custom IP wizard is used to import the newly created IP into the design. Once imported the peripheral is simply connected to the data bus. At this point the bitstream can be generated and programmed onto the FPGA. However without the appropriate software application to control the hardware nothing interesting will happen after programming the FPGA. Software applications written in C are used in order to establish communication between the PowerPC and the FPGA fabric. These applications will be discussed in a later section.

5.4 Required modifications

As previously mentioned modifications to the implementations targeted for simulation needed to be made in order to work correctly on the development board. The crucial change was in regard to reading data from the write FIFO at the appropriate time. This required an $ok\_to\_send\_data$ signal to be added to the interface of all implementations. In order to facilitate the generation of this signal a new state, $read\_fifo\_delay$, was added to the control unit of all implementations. The $send\_hash$ state was removed from all control unit implementations, because these components are no longer at the top level and as a result the full 512-bit hash can be obtained from them without the need to conserve I/O pins. The appropriate connections will be made inside of the peripheral stub code to ensure that the final hash is sent over the data bus accordingly. In addition an $init$ state was added to PHASH to ensure the $ok\_to\_send\_data$ signal is generated correctly when the run signal is asserted. The functionality and structure of all other components remained unchanged. The resulting state diagrams for SHA-512 and Whirlpool are shown in Figure 5.2 and Figure 5.3, respectively. The additional states allow the read from the write FIFO to be initiated one cycle early, so that in the $read\_data$ state the output from the write FIFO will contain valid data.

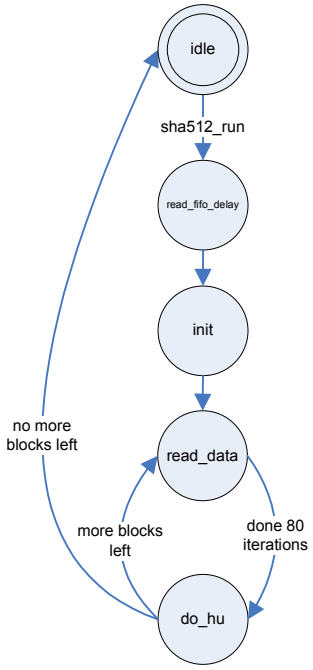

Figure 5.2: *State diagram of the modified SHA-512 control unit for hardware*

The $read\_fifo\_delay$ state in PHASH is not part of the initialization chain and is therefore embedded further in the control unit as shown in Figure 5.4. Its function is identical to the $read\_fifo\_delay$ states in both SHA-512 and Whirlpool.

As previously mentioned, all implementations, excluding PHASH with 2 W cipher instances, used the same custom IP wizard options. The wizard only allows the inclusion of a single write FIFO within a peripheral, however the PHASH implementation using 2 W cipher instances requires two such FIFOs. In order to create a functional implementation, two components were generated using the custom IP wizard. The first was a component for the PHASH implementation, however it did not include a write FIFO. The second component did not contain any hashing function implementations and only contained a single write FIFO. Two instances of this component were placed into the design along with the PHASH component. The appropriate outputs from the write FIFO components were connected to the inputs of the PHASH component and the appropriate outputs from the PHASH component were connected to the inputs of the write FIFOs. This setup enabled multiple write

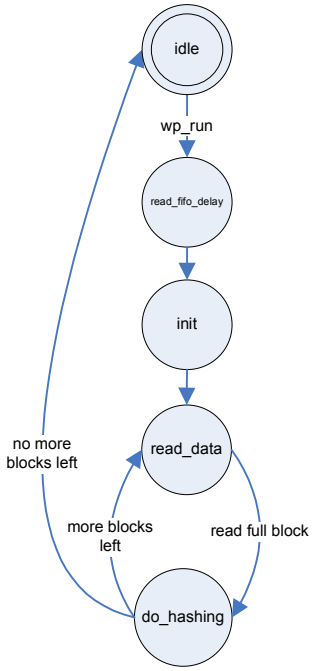

Figure 5.3: *State diagram of the modified Whirlpool control unit for hardware*

FIFOs to provide data to a single peripheral.

5.5 Software applications

In order to communicate with the generated hardware systems, software written in C is used. The software provides an interface for accessing the previously generated system. Simple calls exist for reading and writing to the software addressable registers as well as for writing data to the write FIFOs. In order to access the contents of the CF card, the XilFATFS library needs to be included. This library provides several function calls, resembling the familiar $fopen$, $fread$ and $fwrite$ C functions that are used to access data on the CF card.

Two sets of applications were created for each hashing function. The first application was used to read in pre-padded data located on the CF card. This data will be used to test the functionality of the implementations. The second application did not require using the

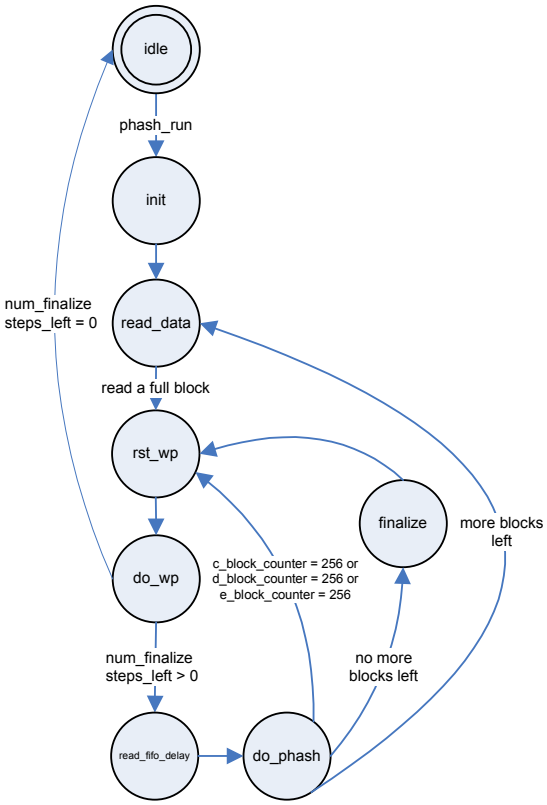

Figure 5.4: *State diagram of the modified PHASH control unit for hardware*

XilFATFS library. It was used to obtain internally generated test data that would also be used to verify the functionality of the implementations. Further details pertaining to the tests will be provided in the next chapter.

Chapter 6

Testing procedures and results

This chapter describes the testing procedures used in order to verify correct functionality of all simulation and hardware implementations. All obtained results are also presented.

Throughout the implementation process many unit tests were performed to verify the functionality of individual components. The correct operation of these components was verified simply by looking at the resulting simulation waveforms. As more components were implemented more complex testing was performed containing several of these components. At this point looking at the simulation waveforms became a tedious process, so several testbenches were created in order to facilitate the integration testing. Testbenches were also used to test the final simulation implementations. To test hardware implementations the actual results obtained were compared to values produced by external C programs.

One of the goals of this work was to compare the SHA-512 and Whirlpool implementations to those currently published in the literature. Yet, another goal was to demonstrate the implementations of SHA-512, Whirlpool and PHASH on a Virtex-4 FX60 FPGA to verify functionality in hardware. Another goal was to determine the maximum possible throughput PHASH can achieve targeted for the state-of-the-art Virtex-5 LX330 FPGA. Post PAR timing results were used to determine the maximum achievable throughput. A comparison of the SHA-512, Whirlpool and PHASH algorithms targeted for the Virtex-5 LX330 FPGA was also performed.

Xilinx ISE 9.2.04i and Xilinx EDK 9.2.02 were used to obtain all results. Xilinx ISE was used to perform synthesis and to obtain post PAR timing information as well as to

generate post PAR simulation models. Various combinations of synthesis, map and PAR options were used in order to obtain the maximum throughput for each hashing function. Xilinx EDK was used in order to create the hybrid FPGA systems to verify functionality of all hashing algorithms in hardware.

6.1 Testing VHDL models

A testbench has been written to test each VHDL model. The diagram depicting the structure of the testbenches used is shown in Figure 6.1.

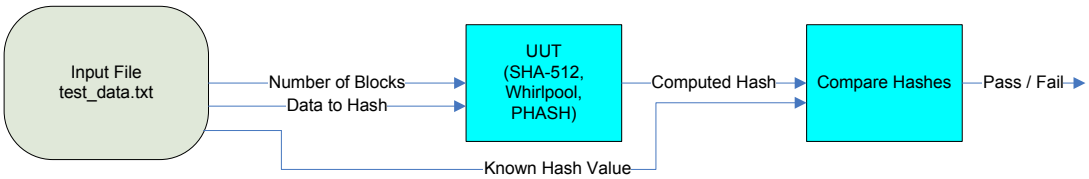

Figure 6.1: *Structure of testbenches used to test SHA-512, Whirlpool and PHASH implementations*

The testbench is responsible for reading pre-padded data to be hashed from a file, sending the data at the appropriate time, as well as comparing the obtained hash to a known value. The input file read by the testbench contains groups of three lines. The first line specifies the number of blocks of data to be hashed. The second line contains the actual pre-padded data to be hashed and the last line contains a known value for the given data. Multiple tests can be included in the same input file. All values are written in hexadecimal notation. The input files were generated using external C programs. These files were then processed by the hashing algorithm in order to verify their correctness.

The SHA-512 and Whirlpool test data consists of 240 tests, starting with 20 bytes of data and incrementing by 20 bytes for each consecutive test, for a total of 4800 bytes. The PHASH test data consists of three separate tests. The first consists of 128 test cases. The first case ensures only a single block is hashed, and each consecutive test hashes an extra block. As a result 128 tests ranging from a single block to 128 blocks are performed. The

second test consists of hashing 256 and 257 data blocks. These block counts represent the transition between the first and second reduction levels. The final test consists of hashing 65536 and 65537 blocks. These block counts represent the transition between the second and third reduction levels.

6.2 Testing hardware implementations

In order to test the hardware implementations several binary data files as well as internally generated data was used. The binary data files varied in size from approximately 5 KB to 105 KB. These files were transferred to the CF card and were accessible by the PowerPC. A software application read the contents of the file and performed message padding on the appropriate data. Next, the padded message was transferred into the write FIFOs on the FPGA. Finally the run signal was asserted and the processing started. Once the $done$ signal was asserted the resulting hash value was read out over the data bus. This hash was output to the terminal and was compared against the hash generated by an external C program.

The internally generated data consisted of three sets of tests. The first two sets consisted of a variable number of 00_x and FF_x bytes in a single data block. The first test consisted of only a single 00_x and FF_x byte, respectively. The other tests added an additional byte, up to a certain point. For SHA-512 a total of 111 tests were done for each case. This number represents the maximum number of bytes in a single data block. Using 112 bytes would require a second block. For Whirlpool 31 tests were done and for PHASH 95 tests were done. The final test set using internally generated data consisted of computing the hash of a variable number of bytes. The first test consisted of a single byte of data and consecutive tests added one additional byte. The arbitrary limit of 2048 bytes was used for all three hashing functions. The procedure for obtaining the resulting hashes of the internally generated data was the same as for the binary data files.

6.3 SHA-512 and Whirlpool comparison

The SHA-512 implementation most comparable to the one in this work is described in [12]. It is a basic implementation targeted for the XC2V2000B-F957 FPGA. It is not unrolled and uses a single BlockRAM to store K_t constants. The SHA-512 implementation in this work was targeted for this FPGA and post PAR timing results were obtained. These results are summarized in Table 6.1.

Implementation	CLB Slices	Frequency (MHz)	Throughput (Mbps)
This work	2242	87.72	1123
McEvoy *et al* [12]	2726	109.03	1329

Table 6.1: *Comparison of SHA-512 implementations for the XC2V2000B-F957 FPGA*

Both implementations are close in terms of performance. The implementation in this work requires less CLB slices, however it also operates at a lower frequency. On the other hand, the implementation in [12] requires a greater number of CLB slices and as a result it operates at a higher frequency. The number of CLB slices required per 1 Mbps of throughput is 1.97 and 2.05 for the implementation in this work and in [12], respectively. This metric shows that both implementations are very close in terms of performance.

The two Whirlpool implementations that are most comparable to the ones in this work are described in [14]. One implementation uses BlockRAM to implement the non-linear stage γ and the other implements it using mini S-boxes placed in distributed RAM. These implementations were all targeted for the Virtex-4 LX100 FPGA and post PAR timing results were obtained. These results are summarized in Table 6.2.

Implementation	CLB Slices	Frequency (MHz)	Throughput (Mbps)
This work (BlockRAM)	5006	128.57	6583
McLoone *et al* [14] (BlockRAM)	4956	91.37	4896
This work (Distributed RAM)	6377	131.89	6753
McLoone *et al* [14] (Distributed RAM)	7507	93.56	4790

Table 6.2: *Comparison of Whirlpool implementations for the Virtex-4 LX100 FPGA*

The BlockRAM implementation in this work achieves better performance over the one presented in [14]. The difference between the number of required CLB slices in the two implementations is negligible. However the implementation in this work operates at a much higher frequency and as a result the throughput is increased. The implementation in this work requires only 0.76 CLB slices per 1 Mbps of throughput, whereas in [14] 1.01 CLB slices are required to achieve 1 Mbps of throughput. According to this metric the implementation in this work is over 30% more efficient than the one in [14].

The distributed RAM implementation in this work also achieves better performance over a similar one in [14]. It requires a considerably lower amount of CLB slices and it operates at a significantly higher frequency. As a result, it also achieves a much higher throughput. The implementation in this work requires only 0.95 CLB slices per 1 Mbps of throughput, whereas in [14] 1.56 CLB slices are required to achieve 1 Mbps of throughput. According to this metric the implementation in this work is over 60% more efficient than the published one.

6.4 SHA-512, Whirlpool and PHASH implementations for the Virtex-4 FX60 FPGA

The SHA-512, Whirlpool and PHASH implementations in this work were also targeted for the Virtex-4 FX60 FPGA. This FPGA was used in order to verify functionality of the algorithms in hardware. Before being used in the actual hardware, post PAR simulations were performed in order to verify functionality. Next, post PAR timing results were obtained to determine the maximum achievable throughput on this FPGA. The post PAR results obtained are shown in Table 6.3.

Table 6.3 includes two SHA-512, six Whirlpool and two PHASH configurations. Only two PHASH configurations were obtained because the Virtex-4 FX60 FPGA does not have sufficient resources for a PHASH implementation using 4 W cipher instances.

Both SHA-512 implementations require approximately the same number of CLB slices

Algorithm	Configuration	CLB Slices	Frequency (MHz)	Throughput (Mbps)
SHA-512	Distributed RAM	2073	106.65	1365
SHA-512	BlockRAM	1917	103.17	1321
Whirlpool	1	6605	122.09	6251
Whirlpool	2	6597	123.50	6323
Whirlpool	3	7327	105.28	5390
Whirlpool	4	7937	112.38	5754
Whirlpool	5	4833	138.96	7115
Whirlpool	6	3914	137.08	7018
PHASH	1 W instance	11010	126.71	11353
PHASH	2 W instances	16901	124.24	22264

Table 6.3: *SHA-512, Whirlpool and PHASH implementations for the Virtex-4 FX60 FPGA*

and operate at approximately the same frequency. As a result the difference between their maximum throughputs is negligible. All Whirlpool implementations require between 4000 and 8000 CLB slices to implement and operate at a frequency range between 105 and 140 MHz. The least efficient implementation still attains a throughput of over 5 Gbps.

The W cipher instances used within PHASH require a considerable amount of resources. This increase in resources enables PHASH to achieve a very high throughput. In order to implement PHASH using such instances high-end and therefore more expensive, FPGAs are required. The reason so many resources are required is because these implementations operate on entire 512-bit data blocks at once. In other words, the data path width is 512 bits. As a result a large amount of data is processed in a single cycle, thereby increasing throughput.

In order to choose the best W cipher implementation the results obtained from the Whirlpool implementations will be used. The W cipher is the core of the Whirlpool hashing algorithm and as a result it requires a majority of FPGA resources to implement. Therefore it was assumed that the most efficient Whirlpool implementation also contains the most efficient W cipher implementation.

Currently there exist two published Whirlpool implementations designed using a smaller data path width. The implementation in [18] is designed using a 64-bit data path width. The

implementation in [3] is designed using an 8-bit data path width. These implementations were optimized for area rather than throughput and as a result require less resources to implement, at the cost of decreased throughput. A summary of these implementations is shown in Table 6.4. The resources required to implement Whirlpool (Configuration 1 as shown in Table 4.1) that uses a 512-bit data path width are also included for comparison.

Data path width (bits)	CLB Slices	Frequency (MHz)	Throughput (Mbps)
8 [3]	376	214.00	81.5
64 [18]	1456	131.00	382
512	6605	122.09	6251

Table 6.4: *Whirlpool implementations using different data path widths*

By doing a purely theoretical analysis it can be seen that the two area optimized implementations would not be suitable to use as a replacement of the 512-bit data path width implementation when aiming for high throughput. In order to achieve the throughput of the implementation in this work approximately 77 instances of the 8-bit data path width and 17 instances of the 64-bit data path width implementations would be required. A total of 28952 and 24752 CLB slices would be required to realize these implementations, respectively.

The PHASH throughput for a given configuration depends on the total amount of data being processed, since there is an additional W cipher invocation every 256 blocks processed. In addition a variable number of W cipher invocations occur during the computation of the final hash. Since PHASH is aimed at achieving high performance it will be assumed that mainly large files will be hashed. As a result the average number of W cipher invocations to process a block tends to 1 if the size of the file is sufficiently large. So all stated PHASH throughputs are based on the processing of a single block of data using a single W cipher instance.

In order to achieve the highest throughput with PHASH, the most efficient W cipher implementation should be used. As a result Configuration 5 of the W cipher in Whirlpool

should be used as it achieves the highest throughput. The reason for such a high throughput is that this implementations utilizes 64 BlockRAMs. This allows for a decrease in CLB slice usage since the non-linear stage γ is now implemented in BlockRAMs as opposed to distributed RAM. Therefore, a PHASH implementation using a single W cipher instance requires 64 BlockRAMs and if two W cipher instances are used 128 BlockRAMs are required. The Virtex-4 FX60 FPGA contains a total of 232 BlockRAMs. A majority of these BlockRAMs were used in order to implement the FIFOs used to buffer data as well as for several other components of the system and as a result less than 128 BlockRAMs were available for general use.

In order to avoid over-mapping the BlockRAMs, a W cipher implementation which does not use any BlockRAMs will be used. This means that Configuration 1 and Configuration 2 are the next best candidates. Since these configurations are very close to one another in terms of performance post PAR timing reports for PHASH using both configurations were obtained. The PHASH results shown in Table 6.3 are for the implementation using Configuration 2 as it was able to achieve a higher throughput than Configuration 1.

Both PHASH implementations require a significantly greater amount of CLB slices to implement than Whirlpool but they also provide a much higher throughput. PHASH with a single W cipher instance achieves a throughput over 11 Gbps. The implementation using two instances scales very well as it achieves almost exactly twice the throughput over the single instance implementation.

In addition to obtaining maximum throughput information for all implementations on this FPGA, the post PAR timing information shows that all implementations can operate at a frequency greater than 100 MHz. This allowed the actual hardware system to operate the data bus at a common frequency of 100 MHz, without the need to operate at the next lowest allowable frequency of 66.67 MHz.

6.5 Hardware verification

A total of 10 hardware systems were created, two for SHA-512, six for Whirlpool and two for PHASH. These correspond to the different configurations possible for each algorithm. The generated bitstreams were then programmed onto the FPGA and the appropriate software application was executed by the embedded PowerPC processor. By comparing the generated output to the expected output the results were identical. This completed the hardware verification procedure.

6.6 PHASH throughput for the Virtex-5 LX330 FPGA

In order to choose the most efficient W cipher implementation to use in PHASH on the Virtex-5 LX330, the same type of analysis was carried out as for the Virtex-4 FX60 FPGA. The post PAR timing results for SHA-512 and Whirlpool are shown in Table 6.5. The SHA-512 results are included for completeness.

Algorithm	Configuration	Registers	LUTs	LUT-FF Pairs	Slices	Usage (%)	Frequency (MHz)	Throughput (Mbps)
SHA-512	DRAM	1495	2876	3023	1039	2	140.37	1797
SHA-512	BRAM	1502	2768	2891	1102	2	142.88	1829
Whirlpool	1	3609	9745	10152	3217	6	156.27	8001
Whirlpool	2	3694	9055	9453	2892	5	162.34	8312
Whirlpool	3	3507	10624	10914	3232	6	153.85	7877
Whirlpool	4	2777	9654	9908	3190	6	161.29	8258
Whirlpool	5	2173	6550	6821	2144	4	131.58	6737
Whirlpool	6	2173	5762	6188	2100	4	135.14	6919

Table 6.5: *SHA-512 and Whirlpool implementations for the Virtex-5 LX330 FPGA*

The device usage information for the Virtex 5 family of FPGAs is more detailed than that of the previous Virtex families. Individual register and LUT usage is provided as well as a count of the LUT-FF pairs. A LUT-FF pair represents one LUT paired with one flip-flop within a slice. The % usage is given in terms of the total number of slices used. It is important to note that a slice in the Virtex-5 family differs from a slice in the Virtex-4

family so as a result these values cannot be compared across these two FPGA families. A Virtex-5 slices contains more resources than a Virtex-4 slice.

In order to achieve the highest throughput with PHASH, the most efficient W cipher implementation should be used. As a result Configuration 2 and 4 are the best candidates. Since these configurations are very close to one another in terms of performance post PAR timing reports for PHASH using both configurations were obtained. The results shown in Table 6.6 are for the implementation using Configuration 2 as it was able to achieve a higher throughput than Configuration 4.

A total of 5 PHASH implementations were generated using 1, 2, 4, 8 and 16 instances of the W cipher, respectively. Post PAR timing results were used to obtain the minimum clock period of each implementation. Using this information the maximum throughput was calculated. A simulation model was also generated for each post PAR implementation to verify its functionality. A summary of the results is shown in Table 6.6.

W cipher instances	Registers	LUTs	LUT-FF Pairs	Slices	Usage (%)	Speedup Factor	Frequency (MHz)	Throughput (Mbps)
1	6696	13445	13770	4469	8	1.00	168.07	15059
2	8152	22194	23105	8031	15	1.92	161.29	28903
4	16491	41620	43141	13537	26	3.69	155.04	55566
8	29670	75389	78782	24427	47	6.70	140.85	100958
16	42362	149018	152402	44344	84	12.13	127.39	182624

Table 6.6: *PHASH implementations for the Virtex-5 LX330 FPGA*

The speedup factor represents the efficiency at which the implementation scales. The throughput obtained with a single W cipher instance is used as the baseline and therefore its speedup factor is 1.00. The maximum theoretical speedup factor would be equal to the number of W cipher instances used.

A graph of maximum frequency versus the number of W cipher instances is shown in Figure 6.2.

The thicker line represents the actual obtained data, and the thinner line represents a logarithmic trend line. This graph allows extrapolation of maximum frequency for implementations using more than 16 instances of the W cipher. Using the obtained trend line a

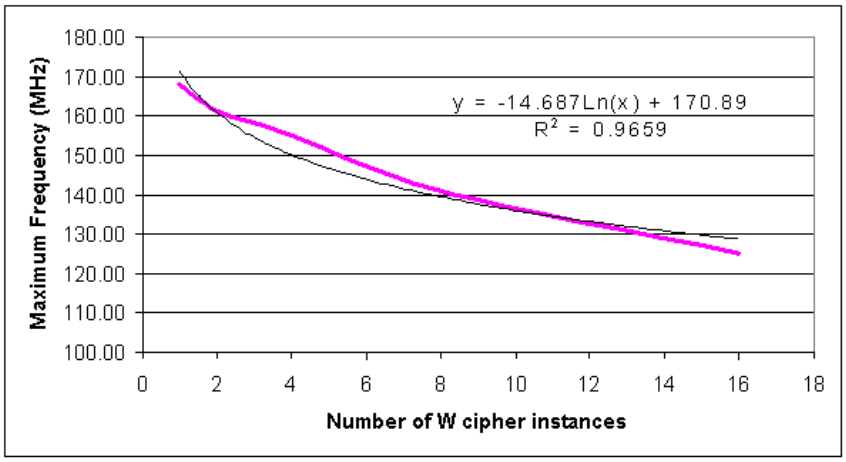

Figure 6.2: *Maximum frequency versus number of W cipher instances in PHASH*

maximum frequency of 120.00 MHz can be computed when using 32 such instances. This frequency translates to a throughput of over 344 Gbps. The data shows that the PHASH algorithm scales very well as the number of instances are increased. Using a single Virtex-5 LX330 FPGA a throughput of over 182 Gbps can be achieved.

A graph of the maximum throughput of all three algorithms targeted for the Virtex-5 LX330 FPGA is shown in Figure 6.3.

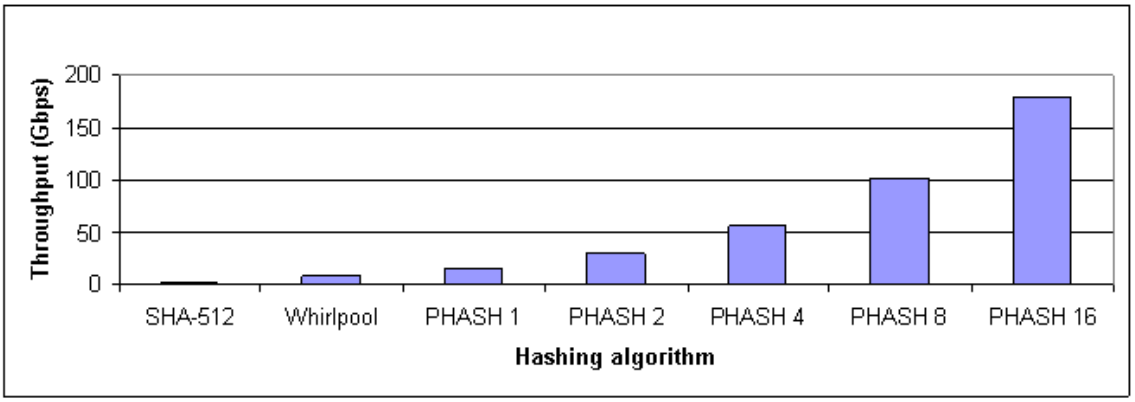

Figure 6.3: *Maximum throughput of SHA-512, Whirlpool and PHASH for the Virtex-5 LX330 FPGA*

The SHA-512 implementation requires 0.58 slices per 1 Mbps of throughput, whereas the Whirlpool implementation only requires 0.35 slices per 1 Mbps of throughput. Not

only is the PHASH implementation more efficient in terms of throughput but also in terms of the number of slices per 1 Mbps of throughput. A PHASH implementation with a single W cipher instance requires only 0.30 slices per 1 Mbps of throughput. The remaining implementations require between 0.28 and 0.24 slices per Mbps. The reason why this ratio decreases as the number of instances increases is because the first W cipher instance in each implementation requires the $data\_mux$ component, whereas the remaining instances do not need it. Increasing the number of instances diminishes the effect of the additional resources required to implement the $data\_mux$ component.

Chapter 7

Conclusions and future work

This work contains a thorough presentation of several implementations of SHA-512 and Whirlpool as published in the literature. Using the information obtained from the literature several SHA-512 and Whirlpool implementations were realized as part of this work. These implementations were compared and contrasted to currently existing ones. In addition, the W cipher configurations in this work were investigated further in order to determine the most suitable one for use in the parallelizable hashing function, PHASH.

The PHASH implementations were targeted for the state-of-the-art Virtex-5 LX330 FPGA in order to demonstrate the maximum achievable throughput of the parallelizable hashing function, PHASH. Post PAR timing information was used to compute the maximum throughput for five PHASH implementations, each using a different number of W cipher instances. It was shown that PHASH can achieve a throughput over 15 Gbps using a single W cipher instance. The PHASH algorithm was able to achieve a throughput over 182 Gbps when using a total of 16 such instances.

Several assumptions were made during the implementation of PHASH in order to facilitate the development process. Future work should include removing most, if not all, of these assumptions in order to create a less constrained implementation. The first step of this process would require the final PHASH draft to be used, as the draft used for the implementation in this work was not finalized. The most noticeable restriction in this implementation of PHASH is that the number of W cipher instances used is required to be a power of 2. If a new implementation were to remove this restriction and allow an arbitrary

number of W cipher instances to be used, several more instances would be able to fit onto the Virtex-5 LX330 FPGA. This would easily push the maximum achievable throughput to over 200 Gbps for this device.

The amount of data the hardware systems in this work are capable of hashing is limited by the amount of BlockRAMs available on the given FPGA. Future works can establish a more robust communication protocol between the PowerPC and the FIFO buffers located on the FPGA in order to accommodate an unlimited amount of data for processing.

None of the implementations in this work include a padder. However by using a hybrid FPGA the padding can be performed in software. Future implementations can include an optional implementation of the padder if a hybrid FPGA is not available in order to provide a more complete system.

The implementation in this work restricts the maximum number of reduction levels in PHASH to three. Future implementations can remove this restriction in order to provide an implementation that is as close to the specification as possible.

Bibliography

[1] I. Ahmad and A. S. Das. Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs. *Computers and Electrical Engineering*, 31(6):345 – 360, 2005.

[2] F. Aisopos, K. Aisopos, D. Schinianakis, H. Michail, and A. P. Kakarountas. A novel high-throughput implementation of a partially unrolled SHA-512. *Proceedings of the Mediterranean Electrotechnical Conference - MELECON*, 2006:61 – 65, 2006.

[3] T. Alho, P. Hämäläinen, M. Hännikäinen, and T. D. Hämäläinen. Compact hardware design of Whirlpool hashing core. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1247–1252, San Jose, CA, USA, 2007. EDA Consortium.

[4] P. Barreto and V. Rijmen. Whirlpool Hash Function. *http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html*, 2006.

[5] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis. Improving SHA-2 hardware implementations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4249 NCS:298 – 310, 2006.

[6] L. Dadda, M. Macchetti, and J. Owen. An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512). *Proceedings of the ACM Great Lakes Symposium on VLSI*, pages 421 – 425, 2004.

[7] FIPS 180-2. *Secure Hash Standard*, August 2002.

[8] International Organization for Standardization. ISO/IEC 10118-3:2003. *http://www.ncits.org/ref-docs/FDIS_10118-3.pdf*.

[9] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott. Comparative analysis of the hardware implementations of hash functions

SHA-1 and SHA-512. *Information Security. 5th International Conference ISC 2002. Proceedings (Lecture Notes in Computer Science Vol.2433)*, pages 75 – 89, 2002.

[10] A. Kaminsky and S. Radziszowski. A Case for Parallelizability of the AHS. preprint, 2007.

[11] P. Kitsos and O. Koufopavlou. Efficient architecture and hardware implementation of the Whirlpool hash function. *IEEE Transactions on Consumer Electronics*, 50(1):208 – 213, 2004.

[12] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. Optimisation of the SHA-2 family of hash functions on FPGAs. *Proceedings - IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures 2006*, 2006:317 – 322, 2006.

[13] M. McLoone and J. V. McCanny. Efficient single-chip implementation of SHA-384 and SHA-512. *2002 IEEE International Conference on Field-Programmable Technology (FPT). Proceedings (Cat. No.02EX603)*, pages 311 – 14, 2002.

[14] M. McLoone, C. McIvor, and A. Savage. High-speed hardware architectures of the Whirlpool hash function. *Proceedings - 2005 IEEE International Conference on Field Programmable Technology*, 2005:147 – 153, 2005.

[15] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[16] NESSIE. New European Schemes for Signatures, Integrity and Encryption. IST-1999-12324. *http://www.cryptonessie.org*.

[17] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard. *http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf*.

[18] N. Pramstaller, C. Rechberger, and V. Rijmen. A compact FPGA implementation of the hash function Whirlpool. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, pages 159 – 166, 2006.

[19] N. Sklavos and O. Koufopavlou. On the hardware implementations of the SHA-2 (256, 384, 512) hash functions. *Proceedings - IEEE International Symposium on Circuits and Systems*, 5:153 – 156, 2003.

[20] W. Stallings. The Whirlpool secure hash function. *Cryptologia*, 30(1):55 – 67, 2006.