IMPLEMENTATION AND BENCHMARKING OF PADDING UNITS AND HMAC FOR SHA-3 CANDIDATES IN FPGAS AND ASICS

by

Ambarish Vyas

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

Committee:

KGaj
Dr. Kris Gaj, Thesis Director

Dr. Jens-Peter Kaps, Committee Member

Dr. Bernd-Peter Paris, Committee Member

Dr. Andre Manitius, Department Chair
of Electrical and Computer Engineering

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

Date: 12/9/2011

Fall Semester 2011
George Mason University
Fairfax, VA

Implementation and Benchmarking of Padding Units and HMAC for SHA-3 Candidates in FPGAs and ASICs

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

By

Ambarish Vyas
Bachelor of Science
University of Pune, 2009

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

Fall Semester 2011
George Mason University
Fairfax, VA

Acknowledgments

I would like to use this oppurtunity to thank the people who have supported me throughout my thesis. First and foremost my advisor Dr.Kris Gaj, without his zeal, his motivation, his patience, his confidence in me, his humility, his diverse knowledge, and his great efforts this thesis wouldn't be possible. It is difficult to exaggerate my gratitude towards him. I also thank Ekawat Homsirikamol for his contributions to this project. He has significantly contributed to the designs and implementations of the architectures. Additionally, I am indebted to my student colleagues in CERG for providing a fun environment to learn and giving invaluable tips and support.

Lastly, and most importantly, I wish to thank my parents,Madhu Vyas and Dinesh Vyas. They have supported me, taught me, educated me and loved me. Despite the distance they were always nearby. My father was always confident about what I was doing. Inspite of him having no idea what I was doing he would still ask every week to explain what my thesis was about. And I don't have words to describe my mother's emotional and moral support. To them I dedicate this thesis.

Table of Contents

List of Tables

List of Figures

Abstract

IMPLEMENTATION AND BENCHMARKING OF PADDING UNITS AND HMAC FOR SHA-3 CANDIDATES IN FPGAS AND ASICS

Ambarish Vyas, M.S.

George Mason University, 2011

Thesis Director: Dr. Kris Gaj

In 2005, a major security flaw was discovered in Secure Hash Algorithm-1 (SHA-1), an NSA-designed cryptographic hash function, standardized by National Institute of Science and Technology (NIST) since 1995. Basic components in the more recent NIST standard SHA-2, introduced in 2002, are quite similar to SHA-1. As both functions are quite similar, it is prudent to expect that the equivalent attacks can be found against SHA-2 in the future. In retort to this possibility, NIST established a contest in search of a new cryptographic hash function family called SHA-3. Presently, the competition is in Round 3 evaluations, with 5 finalists shortlisted out of the 14 from Round 2. Various research groups from the cryptographic community are evaluating the performance of the finalists in hardware while trying their best to be fair in their design decisions. One of the topic of debate in the cryptographic community is whether padding should be included in hardware design or should it be done externally in software and not taken in consideration while evaluating the designs. We propose that padding should be included in the designs for fair evaluations, but should be designed intelligently so that the overall Throughput/Area ratio is not affected by an undesirable amount.

In this thesis, we design and implement padding units for 5 Round 3 SHA 3 finalists for two hardware platforms, FPGAs and ASICs. We show that the worst effect of padding unit on the performance of the candidates does not exceed 18% in FPGAs and the overall ranking of the finalists does not change from the ranking derived from the architectures which do not support padding. Universal padding unit supporting all finalists and SHA-2 was designed for ASICs and the maximum area overhead due to the inclusion of a padding unit is around 9% with no effect on maximum clock frequency. This thesis also focuses on designing a Hash-based Message Authentication Code (HMAC) wrapper for all the SHA-3 finalists and SHA-2.

Chapter 1: Introduction

1.1 Background

Growth in technology and world wide web has made the world smaller and the need to protect information stored electronically very crucial. Network security is thus very important and so cryptographic functions need to evolve for secure transmission of data. Thus hash functions are of great significance in the modern era. Compared to non-cryptographic hash functions, cryptographic hash functions are much more heavy in terms of computations and need more resources and execution time. Therefore they are used only where the protection against security attack is necessary. Cryptographic hash functions takes message as input and gives an output of fixed length called *Message Digest* or *Hash Value*. Combined with other cryptographic transformations, such as secret-key and public-key ciphers, hash functions can be used to provide message integrity, authentication, and non-repudiation [1].

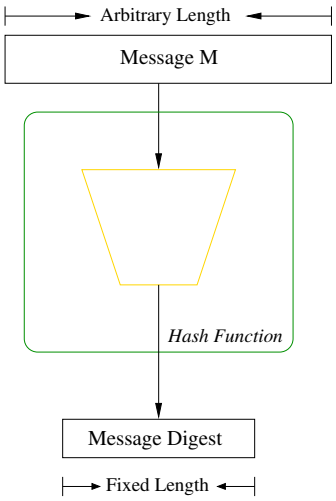


Figure 1.1: Simplified block diagram description of Hash Function

Hash functions have three basic properties :

1. **Compression** : It should be easy to compute the output of fixed length, from an arbitrary length input. For a given input *'M'*, one should be able to compute *H(M)* easily with fixed length output.

2. **Collision resistance** : It should be computationally infeasible to find any two messages which yield the same hash output.
   For given inputs *'M_1'* and *'M_2'* : $H(M_1) \neq H(M_2)$

3. **Preimage resistance** : It should be computationally infeasible to compute the input from the obtained output.
   For a given X, it is not possible to find 'M' such that *H(M)* = X.

Fig 1.1 shows simplified block diagram of a hash function. Message *'M'* of arbitrary length is input to the function and finally a fixed length output is obtained [1] [2].

1.1.1 Applications

1. **Pseudorandom generation**: Hash functions can be used in the generation of pseudo-random output. In hash functions there is no correlation between input and output bits and even a single bit change can change around 50% of the output bits.

2. **Fingerprint of a program**: Hash functions can be used to generate fingerprints of a document or a program which can help to detect a modification by a virus or intruder.

3. **Storing Passwords**: Instead of storing the ID and the password of a user in plaintex, the system can store ID and hash value of the password. In order to authenticate a user, the password presented by the user is hashed and compared with the stored hash value.

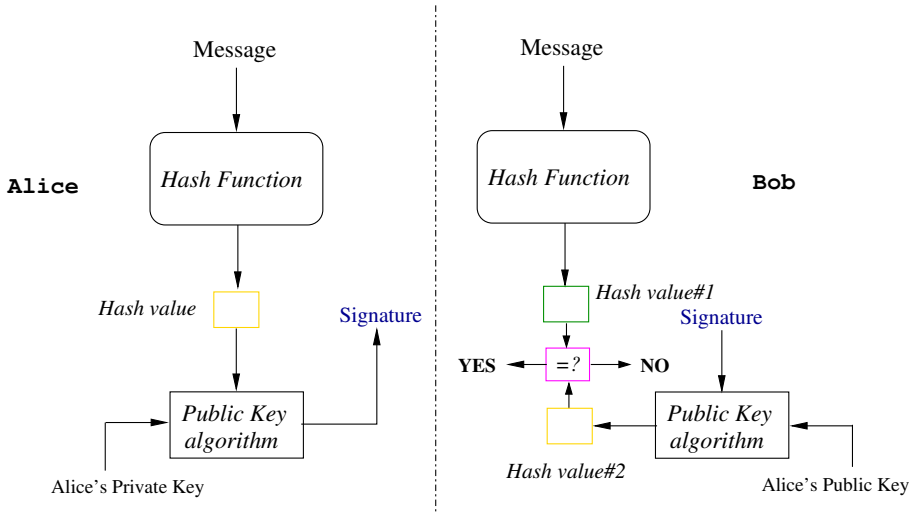


Figure 1.2: Hash Function used for Digital signature generation

4. **Digital Signatures**: Most widely used application of cryptographic hash functions.

   Digital signatures are used for verification of message integrity and strong authentication of message sender. To establish whether any changes have been made to a message, **Message digests** calculated before, and after the transmission of a message can be compared [3]. Fig 1.2 gives an example of verifying message integrity. ***Alice*** sends a message to ***Bob*** along with a digital signature. Digital signature is calculated by first taking the hash of the message and then encrypting it using **Alice's private key**. On the other side, ***Bob*** also calculates a hash value using the same Hash function and compares it with the decrypted signature as shown in Fig 1.2. If both hash value match then the message has not been altered [4].

1.1.2 Padding

The foremost use of padding in classical ciphers was to prevent attacks from cryptanalysts by reducing the predictability of the message. Predictability lies generally at the start and end of a message, with greetings or subject.

One of the infamous examples of padding in classical ciphers, which caused a false impression and led to Admiral William Hasley, Jr. to drop of his pursuit of a Japanese task force in the Battle of Leyte gulf [5]. A padding string was added at the start and end of the message by Admiral Nimitz troop to avoid japanese cryptanalysts to guess the predictable phrases at the start and end. Admiral Nimitz's clerk used the phrase "The World Wonders" [5] at the end of the message. While deciphering the cipher text, Hasley's officer removed the leading phrase, but thought the end string looked appropriate and a part of the message. So Hasley read the message as, *"Where is, repeat, Where is TASK Force Thirty Four?The world wonders"* This created an resentment and Hasley found it a harsh statement on his pursuit, and returned back in anger.

Majority of the cryptographic hash functions process messages divided in fixed length blocks. But we cannot restrict a message to be a multiple of a fixed-length block, this lead to inclusion of padding schemes in the hash functions. The job of the padding rule is not just to extend the length of the message and make it possible for the hash functions to process the total message, but it is also critical regarding security of the hash functions. It is important that the padding rule preserves the collision resistance of the underlying compression function. A padding rule should be injective [6], because an injective function preserves distinctiveness. Injective function can also be called as one to one function as every element in the codomain of the function is mapped to at the most one element from its domain. Padding should be reversible that means it must be possible to determine the original message from the padded message [7].

Merkle padding rule [4] is one of the most commonly used padding rules in hash functions. The Padding rule is defined as,

$$PAD(M) = M||1||0_d||Len(M)_s$$

where

M is the Message,

0_d is a string of d zeros

d is the smallest non-negative integer required,

Len(M)$_s$ is length of the Message M in bits, encoded using s bits

s is generally chosen as 64 or 128.

This padding rule is used in the Merkle-Damgard construction of hash function [4]. Drawback of Merkle padding rule is that it can only support message of length 2^s-1. Present NIST standard SHA-2 [8] uses Merkle padding rule.

1.2 Previous work

As of writing this thesis, to the best of my knowledge there are very few published implementations of padding in hardware for the SHA-3 finalists. Until now, only [9,10] provide results and evaluations of padding circuit in hardware. [11] has very comprehensive evaluation of effect of padding, and has two versions for the padding unit.

1. Assumes that the message size is a multiple of a word (32 bits),

2. Does not put any restriction on the message size.

Both versions are extreme cases, where one is overly optimistic and the other is too pessimistic. The results show that the circuit slows down the overall performance considerably, which should not be the case if the design is done properly. Jungk et al. in [10] sadly do not give out much details and do not have comprehensive performance analysis and reports on affect of padding in hardware.

Regarding HMAC implementations, to the best of my knowledge there is no previous work regarding HMAC implementation on SHA-3 candidates. Mostly HMAC implementations [12, 13] are on SHA-1 [14] and MD5 [15]. The strength of the HMAC depends on the hash function deployed, and as SHA-1 and MD5 have been found to be vulnerable against security attacks, the HMAC implementations are in the same arena of security holes. NIST commented on the cryptanalysis and said that the HMAC is not under the same security attack as SHA-1 but recommended moving to SHA-2. However, I could locate only one implementation of HMAC based on SHA-2 [16] . The SHA-2 implementation is comprehensive and have in depth analysis of energy, throughout, throughput/area criteria. Results generated are for Virtex-2 and Virtex-E which are old devices and do not have as many resources as the modern FPGAs.

Chapter 2: Design Methodology

This thesis follows the same established ground rules as proposed by [17] [18].
These decisions are listed below:

1. **Hardware Platform**: FPGAs as primary implementation platform. Only Configurable Logic Blocks (CLBs) in Xilinx FPGAs, and Logic Elements or Adaptive Look-Up Tables (ALUTs) in Altera FPGAs are used for synthesis and implementation. No Block RAMs, DSP units, or multipliers are used.
   Universal Padding Unit was implemented on the ASIC platform. ASIC was based on 65nm Standard cell CMOS technology.

2. **Interface and protocol** : Uniform input/output interface is used in implementations and is same as proposed by [19].

3. **Language** : VHDL.

4. **CAD tools** :

   - **FPGA**
     - Xilinx - Xilinx ISE Design Suite v 13.1.
     - Altera - Quartus II v 11.1 Subscription Edition Software.

     Open source benchmarking tool, called ATHENa (Automated Tool for Hardware EvaluatioN), developed at George Mason University is used for benchmarking of source codes and optimization of FPGA tool options [20].

   - **ASIC**
     - Front-end tool : Synopsys Design Compiler D-2010.03-SP1-1

– Back-end tool : Cadence Design Systems Encounter Digital Implementation v10.12-s181_1

5. **Optimization target** : Throughput/area.

The source codes of the hash cores are taken from [21] and the details regarding the implementation were found at [22]. Universal testbench developed by CERG-GMU is used to verify all of the SHA-3 Round 3 finalists and SHA-2. The testbench accepts test vectors generated by the script developed in Perl. The script uses the Known Answer Test (KAT) test vector files available as a part of each candidate's submission package.

2.1 Interface and Protocol

This section provides information about the Interface and Protocol used for SHA-3 Round 3 finalists and SHA-2 and is similar to what is proposed by [17].

2.1.1 Interface

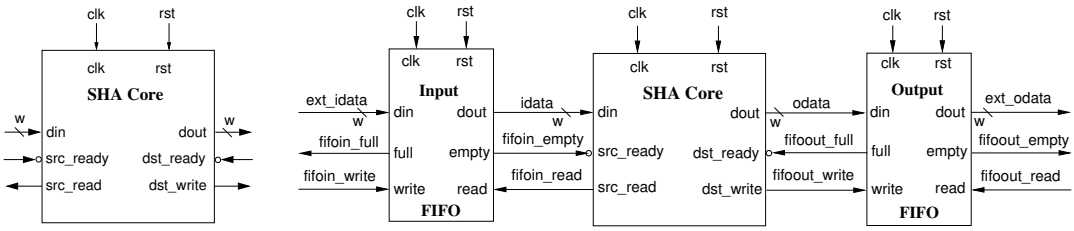
Table 2.1: Description of Interface Signals

Signal	Direction	Description
din	Input	'w' bits wide input data bus.
src_ready	Input	**Active low** source ready indicator.
src_read	Output	Set high to read next word from source.
dst_write	Output	Set high to write next word to destination.
dst_ready	Input	**Active low** destination ready indicator.
dout	Output	'w' bits wide output data bus.
rst	Input	Synchronous **Active high** global reset.
clk	Input	Rising Edge Triggered global clock.

Table. 2.1 briefly describes function, width and names of the input and output ports in the SHA interface as shown in Fig. 2.1(a). Parameter 'w' is the in data bus width depending on the function. To be specific, w= 32 for SHA-256 and w=64 for the remaining functions. Active high *rst* signal resets the SHA core, which now becomes ready for a new message. *src_ready* is an input used by the source of data to indicate to the SHA core that the next

word of data is ready. *src_read* is an output used by the SHA core to read data from the source. *dst_ready* signal indicates that the output destination is ready and can be written to by the SHA core. *dst_write* control signal is an output of the SHA core, indicating to the destination that SHA core wants to write data to it.

The configuration used to test the SHA core is as shown in Fig. 2.1(b) SHA core is assumed to be surrounded by two standard FIFO modules: Input FIFO and Output FIFO. Each FIFO module generates signals fifo_empty and fifo_full indicating that the FIFO is empty or full, respectively. Each FIFO accepts control signals fifo_write and fifo_read, indicating that the FIFO is being written to and read from, respectively. The FIFOs are not a part of the SHA core and are used just to test the SHA core. Thus, during verification of the SHA core, the FIFOs can be implemented as a part of a testbench and for benchmarking they are not taken into account.



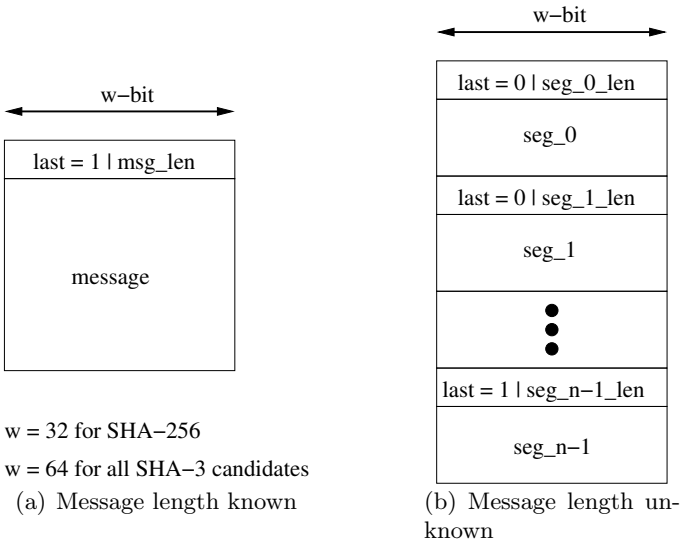
(a) Input-Output Interface (b) FIFO configuration

Figure 2.1: Interface of SHA core and a typical configuration with surrounding input and output FIFOs.

2.1.2 Protocol

A protocol describing a format of input is shown in Fig. 2.2, In the simplest case, the input consists of the length of the message in bits followed by the message itself. The message length is the actual message length before padding. Protocol shown in Fig. 2.2(a) can be used when length of the message to be hashed is known in advance and the length can be represented in 'w' bits. Protocol in Fig. 2.2(b) is used when the length of the message is not known from the start, or is greater than what can be represented in 'w-1' bits (2^w).

So the message is divided into known length segments. The lengths of segments from seg_0 to seg_n-2 are required to be in the multiples of the message block length corresponding to the algorithm. Also when the message is bigger than what can be represented by 'w-1' bits ($2^(w-1)$-1), it can be split into segments and the latter protocol (b) can be used. Protocol (a) is actually a subset of the protocol (b), because situation (a) can be seen as concerning message which consists of only one segment. *last* is a one bit flag which signifies the last segment of a message or just one-segment message in case of situation (a).



(a) Message length known

(b) Message length unknown

Figure 2.2: Input data format for two different operation situations. Notation: msg_len= message length before padding, seg_i_len= segment i length before padding, seg_i= the ith segment of the message, '|' concatenation.

Chapter 3: Padding Rules

This section describes padding rules of all 5 SHA-3 finalists and SHA-2.

Table 3.1: Overview of Padding Schemes of 5 Candidates and SHA-2.('||' stands for concatenation

Sr. No.	Algorithm	Padding Scheme
1.	**BLAKE-256**	M ‖ '1' ‖ "000...000" ‖ '1' ‖ (Message Length)$_{64}$
	BLAKE-512	M ‖ '1' ‖ "000...000" ‖ '1' ‖ (Message Length)$_{128}$
2.	**Grøstl**	M ‖ '1' ‖ "000000" ‖ (Number of Blocks)$_{64}$
3.	**Keccak**	M ‖ '1' ‖ "000000" ‖ '1'
4.	**JH**	M ‖ '1' ‖ "000...000" ‖ (Message Length)$_{128}$
5.	**Skein**	If Message length is a Multiple of a byte : M ‖ "000...000" Else : M ‖ '1' ‖ "000...000"
6.	**SHA-256**	M ‖ '1' ‖ "000...000" ‖ (Message Length)$_{64}$
	SHA-512	M ‖ '1' ‖ "000...000" ‖ (Message Length)$_{128}$

Brief description of padding scheme related to particular hash function is shown in Table. 3.1. BLAKE and SHA-2 have varying schemes for 256 and 512-bit variants.

3.1 BLAKE

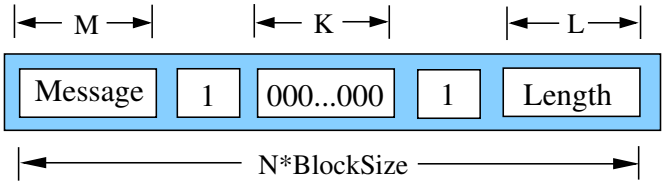


Figure 3.1: BLAKE Padding Scheme

Fig. 3.1 shows the padding rule of BLAKE. Suppose a Message 'M' bits long. First the input message is appended with '1' at the end of the input message, followed by minimum

required ‘K’ bits of zeroes. Then to this sequence ‘1’ is appended, followed by message length of ‘L’ bits such that the total message size after padding is a multiple of the block size.

Minimum number of bits appended are (L+2).

Maximum number of bits appended are (Blocksize+L+1).

$$HashSize = 256 : L = 64, Blocksize = 512, M + K + 2 \equiv 448 \bmod 512.$$

$$HashSize = 512 : L = 128, Blocksize = 1024, M + K + 2 \equiv 896 \bmod 1024.$$

3.2 Grøstl

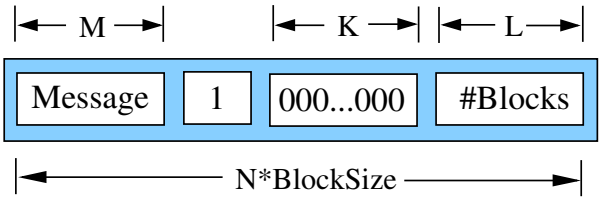


Figure 3.2: Grøstl Padding Scheme

Fig. 3.2 shows the padding rule for Grøstl. Suppose a Message ‘M’ bits long. First the input message is appended with ‘1’ at the end of the input message, followed by minimum required ‘K’ bits of zeroes. Then to this sequence, number of blocks of message after padding represented by ‘L’ bits is appended at the end such that the total message size after padding is a multiple of the block size.

Minimum number of bits appended are (L+1).

Maximum number of bits appended are (Blocksize+L).

$$HashSize = 256 : L = 64, Blocksize = 512, M + K + 1 \equiv 448 \bmod 512,$$

$$HashSize = 512 : L = 64, Blocksize = 1024, M + K + 1 \equiv 896 \bmod 1024.$$

3.3 JH

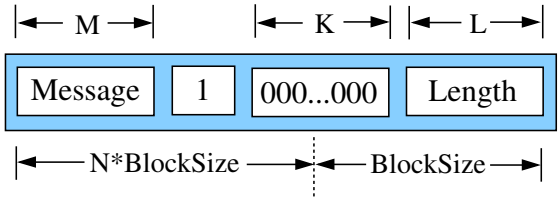


Figure 3.3: JH Padding Scheme

Fig. 3.3 shows the padding rule of JH. Suppose a Message of length 'M' bits long. First the message is appended with '1' at the end of the message, followed by minimum required 'K' bits of zeroes. Then to this sequence, message length of 'L' bits is appended at the end such that the total message size after padding is a multiple of the block size.
Minimum number of bits appended are Blocksize.
Maximum number if bits appended are (2*Blocksize-1) JH will always contain an extra block with no message bits in it.

$$HashSize = 256/512 : Blocksize = 512, K = 383 + (512 - M) \bmod 512.$$

3.4 Keccak

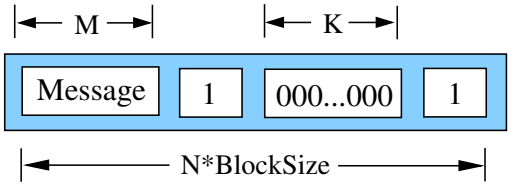


Figure 3.4: Keccak Padding Scheme

Fig. 3.4 shows the padding rule of Keccak. Suppose a Message of'M' bits long. First the

message is appended with '1' at the end of the message, followed by minimum required 'K' bits of zeroes. Then to this sequence '1' is appended such that the total message size after padding is a multiple of a block size.

Minimum number of bits appended are 2.

Maximum number of bits appended are (Blocksize+1).

$$HashSize = 256 : Blocksize = 1088, (M + K + 2) \bmod 1088 = 0$$

$$HashSize = 512 : Blocksize = 576, (M + K + 2) \bmod 576 = 0$$

3.5 Skein

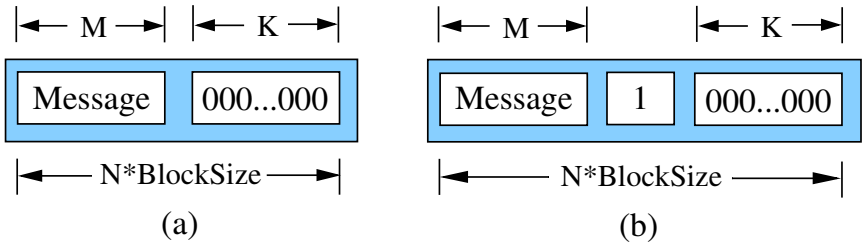


Figure 3.5: Skein Padding Scheme.(a) shows the padding rule when the message length is a multiple of a byte and (b) shows the padding rule when the message length is not a multiple of a byte.

Fig. 3.5 shows the padding rule of Skein. Suppose a Message of 'M' bits long. If 'M' is a multiple of a byte, then the message is appended with minimum required 'K' bits of zeroes such that the total size after padding is a multiple of the block size. If 'M' is not a multiple of a byte, then '1' is appended at the end of the message and followed by minimum required 'k' bits of zeroes. Skein padding rule will never result in an extra block.

If 'M' is a multiple of a byte,

Minimum number of bits appended are 0.

Maximum number of bits appended are (Blocksize-8).

Else,

Minimum number of bits appended are 1.

Maximum number of bits appended are (Blocksize-1).

$$HashSize = 256/512 : Blocksize = 512$$

$$M \bmod 8 = 0 : (M + K) \bmod 512 = 0$$

$$M \bmod 8 \neq 0 : (M + K + 1) \bmod 512 = 0$$

3.6 SHA-2

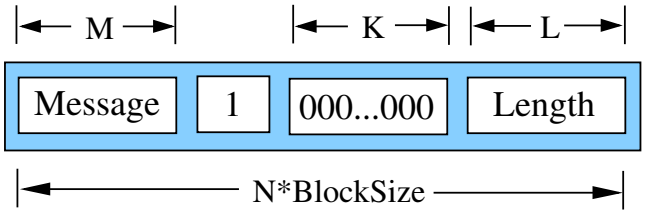


Figure 3.6: SHA-2 Padding Scheme

Fig. 3.6 shows the padding rule of SHA-2. Suppose a Message'M' bits long. First the message is appended with '1' at the end of the message, followed by 'K' bits of zeroes. Then to this sequence, message length represented by 'L' bits is appended at the end such that the total message size after padding is a multiple of the block size.

Minimum number of bits appended are (L+1).

Maximum number of bits appended are (Blocksize+L).

$$HashSize = 256 : L = 64, Blocksize = 512, M + K + 1 \equiv 448 \bmod 512$$

$$HashSize = 512 : L = 128, Blocksize = 1024, M + K + 1 \equiv 896 \bmod 1024$$

Chapter 4: Padding Unit

4.1 General Block

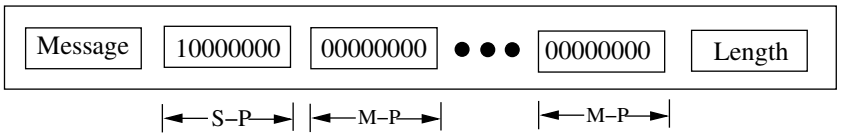


Figure 4.1: Padding scheme of SHA-2 showing S-P: Start Padding and M-P: Mid Padding.

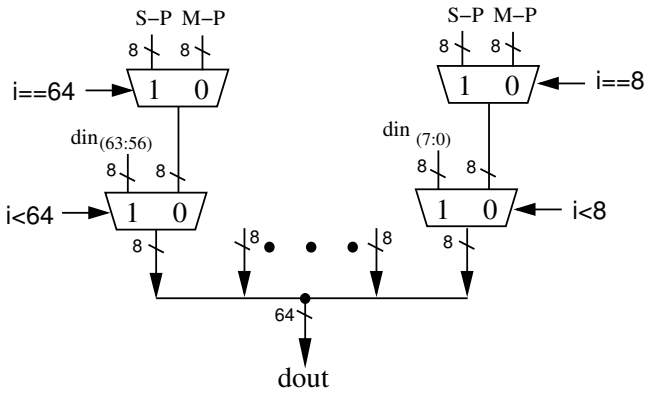


Figure 4.2: SHA-2 Padding unit using comparators. S-P : Start Pad String, M-P: Mid Pad String, End Pad String. 'i' is the number of padding bits in the last word of the message

Fig. 4.1 shows in detail the padding strings when the message size is a multiple of a byte. Input message word is divided into equal segments of a byte each. The padding block consists of two levels of multiplexers. Each two level MUX combination has a constant value associated with it, starting from 64 and ending with 8. They either select the input message byte or the respective padding string to pass. Padding string is either *Start padding* or *Mid padding*, and in some cases there is *End padding string*. These selections are done

by two select signals as shown in Fig. 4.2. 'i' is calculated using the message length. If the position is less than a constant value (multiple of 8), you let input message segment pass. If the constant is equal to i, then the Start Padding String, S-P, is selected. Depending on the padding scheme of an algorithm the values of S-P and M-P are determined. When you want to pass just the padding string for the whole word then 'i' should be greater than the highest constant value which in the figure shown above is 64. Which means any value greater than 64 for 'i' will pass M-P through to the output. So 'i' is 7 bits wide which means we need eight 7-bit comparators. Fig. 4.3 shows three possible cases and values of 'i' during each case. 'i' is the number of padding bits in the last word of the message. Case 3 is a special case where we want only zeros to pass to the core.

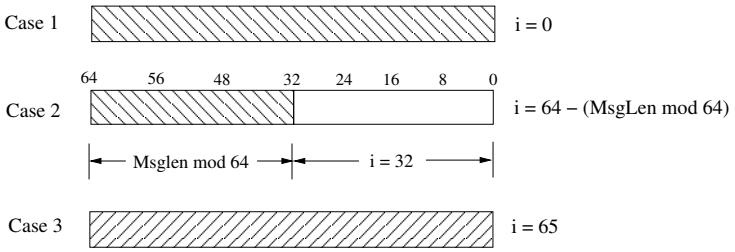


Figure 4.3: Message divided into three cases showing word before, during and after padding

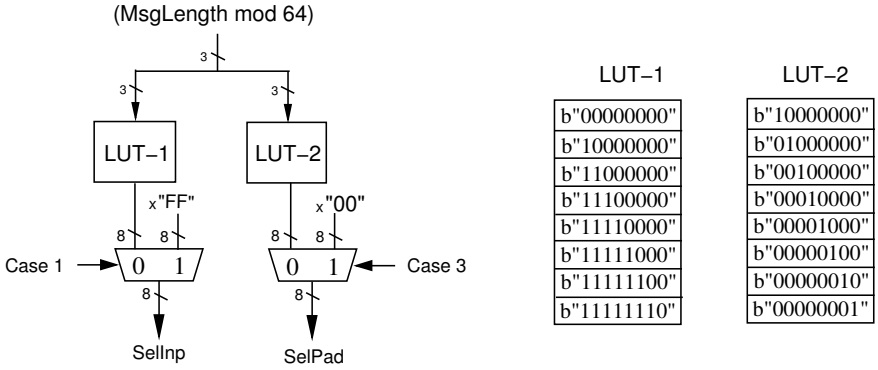


Figure 4.4: Block Diagram for unit which calcualtes select signals for the padding unit logic.

The more efficient way to calculate the select signals for the mux is by using a decoder. The decoders for the two select signals are implemented using Look up tables as shown in

Fig. 4.4 . We assume the message ends on the boundary of a BYTE, thus we can ignore the first three bits of the message length. For input word size 64 bits, bit position 5 to 3 is used as the address to the look up tables. If the input word size is 32 bits (in SHA-256) the input address is just 2 bits and output of the lookup table is 4 bits as shown in Fig. 4.6. To pass the input message 'SelInp' is set to all '1's. When it is required to pad with '0's 'SelInp' is set to "00".

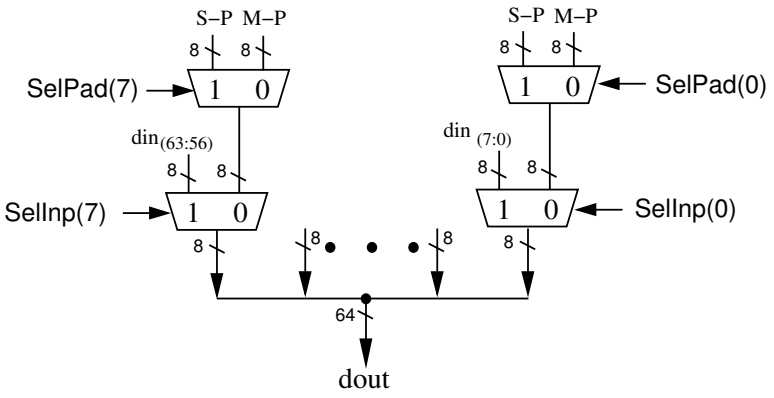


Figure 4.5: SHA-2 Padding unit using decoder logic. S-P : Start Pad String, M-P: Mid Pad String, End Pad String. 'i' is the number of padding bits in the last word of the message

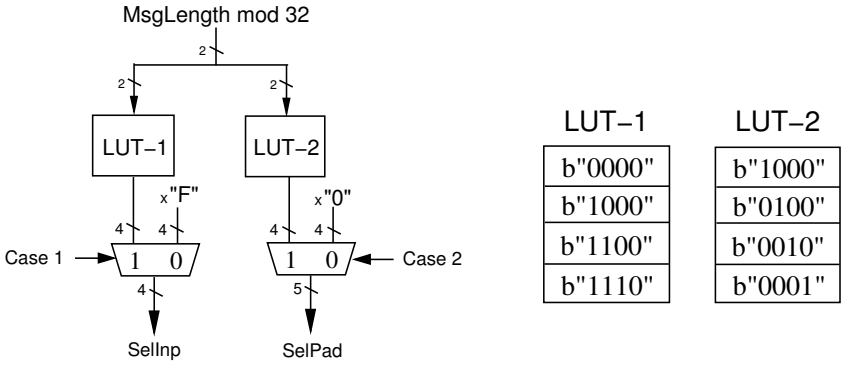


Figure 4.6: Block Diagram for unit which calcualtes select signals for the padding unit logic for word size 32 bits.

Table 4.1: Area and maximum clock frequency results for both version of general padding unit on Xilinx FPGAs

		Virtex 5		**Virtex 6**	
	Type	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [CLB slices]	**Max.Clock** [MHz]
Padding Unit	Comparator	38	294.72	34	321.54
	Decoder	32	313.97	31	330.25

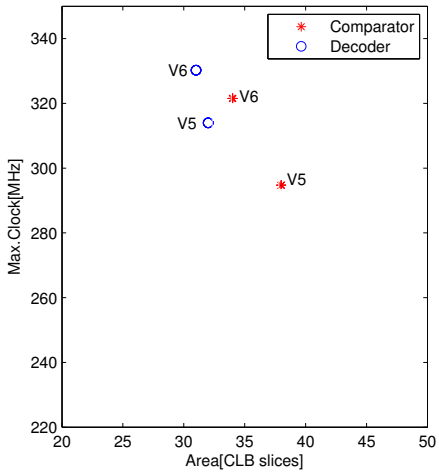


Figure 4.7: Maximum clock frequnecy vs. Area of general padding for both versions in two xilinx families. V5- Virtex 5 and V6- Virtex 6.

As shown in Table. 4.1 padding unit which uses decoder logic to calculate the select signals is faster and slightly smaller. Moreover the controller does not have to calculate the position value also, which makes the controller smaller too compared to the one which uses comparators. Fig. 4.7 plots the improvement of area as well as frequency when decoders are implemented using LUT instead of using comparators.

4.1.1 Multi-Stage Adder

As shown in Table. 3.1 SHA-2, BLAKE, JH appends message length in bits at the end of the message and Grøstl appends message length in multiples of block size. Table. 4.2 lists the length of the counter needed by the algorithms. In some algorithm the counter size changes depending on the variant.

Table 4.2: Counter Size used in the the Padding rule

Algorithm	Variant	Counter Size
BLAKE	256	64-bit
	512	128-bit
JH	256	128-bit
	512	128-bit
Grøstl	256	64-bit
	512	64-bit
SHA-2	256	64-bit
	512	128-bit

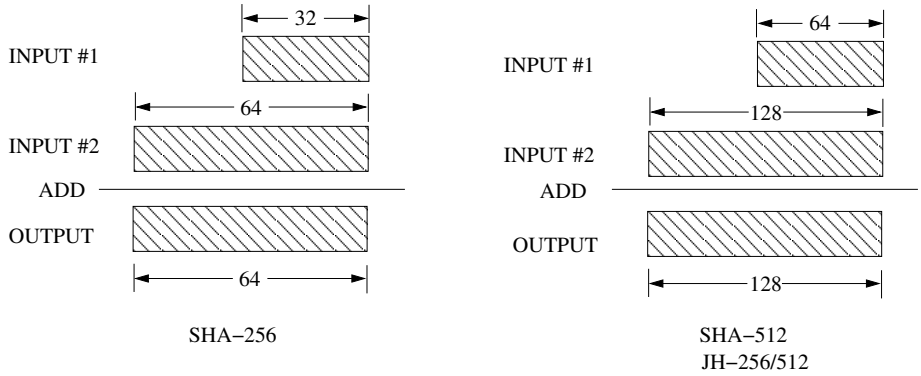


Figure 4.8: Adder configuration used to calculate the size of the message for SHA-2 and JH

JH and SHA2 have a very short critical path and such a wide adder is certain to increase the critical path due to the carry chain propagation and thus will decrease the high performance of the algorithm. This is avoided by using a Multi-Stage adder [23]. Carry signals are propagated in subsequent clock cycles to the next adder. In the Fig. 4.9 each adder is 16-bits wide and the registers are 17-bit wide except the last one as we ignore

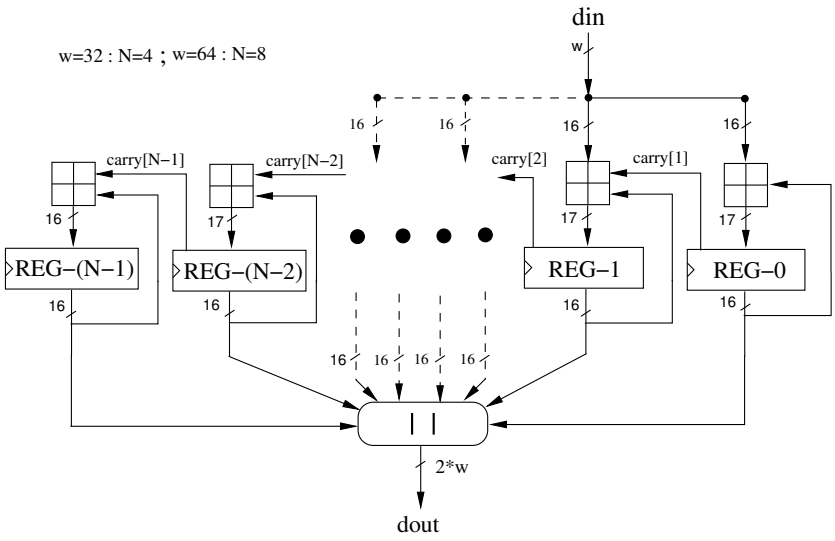


Figure 4.9: Block Diagram of Multi Stage Adder

the carry. Depending on the output the number of registers and adders change. The output is available in 8 clock cycles for 128-bit output and in 4 clock cycles for 64-bit output. Input is divided in segments of 16 and fed into adders. The other input is taken from the register where result of previous addition was stored. The outputs of the registers are concatenated to get the final output.

Fig. 4.8 show the two different cases the adders are used. The message segment length is added to the previous stored length to get the total length of the message.

Table 4.3: Area and maximum clock frequency results for combinational standard carry chain adder ('+' in VHDL)

	Virtex 5		**Virtex 6**		**Stratix III**		**Stratix IV**	
Adder size	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [ALUTs]	**Max.Clock** [MHz]	**Area** [ALUTs]	**Max.Clock** [MHz]
64-bits	32	320.20	31	281.45	65	350.26	65	507.25
128-bits	48	229.30	47	221.236	129	209.69	129	321.75

Table 4.4: Area and maximum clock frequency results for Multi-Stage adder

	Virtex 5		Virtex 6		Stratix III		Stratix IV	
Adder size	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [ALUTs]	**Max.Clock** [MHz]	**Area** [ALUTs]	**Max.Clock** [MHz]
64-bits	54	453.92	51	434.02	74	593.12	74	706.21
128-bits	92	420.875	74	446.43	146	609.76	146	658.76

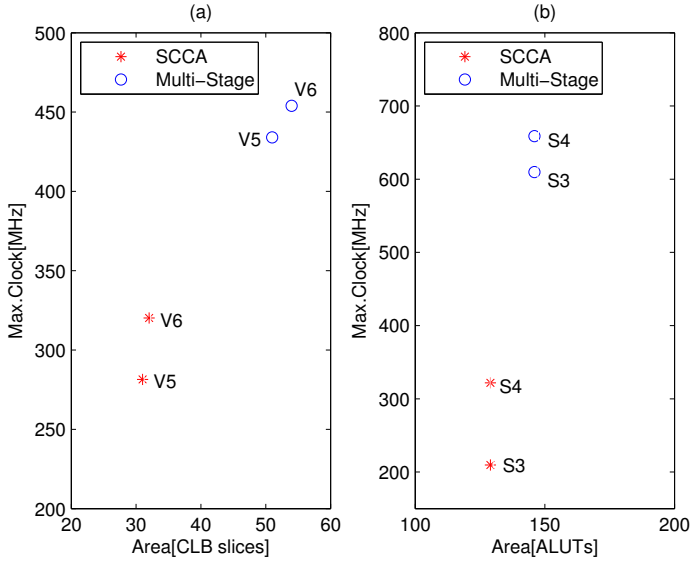


Figure 4.10: Maximum clock frequnecy vs. Area for both versions of adders on 2 xilinx families and 2 Altera families.(a) plot for 64-bit adder,(b) plot for 128-bit adder. V5- Virtex 5, V6- Virtex 6, S3- Stratix III, and S4- Stratix 4

Table. 4.3 and Table. 4.4 gives area and maximum clock frequency of a 64 and 128-bit standard carry chain adder and the multistage adder respectively. Multistage adder is slightly bigger but is aroound 40% faster compared to the basic carry chain adder. Fig. 4.10 shows the improvement in speed with a little increase in area of Multi-stage adder over standard carry chain adders for both 64-bit and 128-bit adders.

4.2 BLAKE

4.2.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

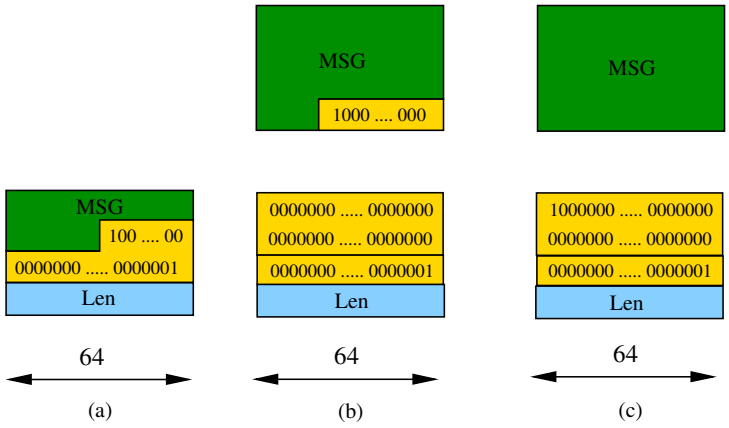


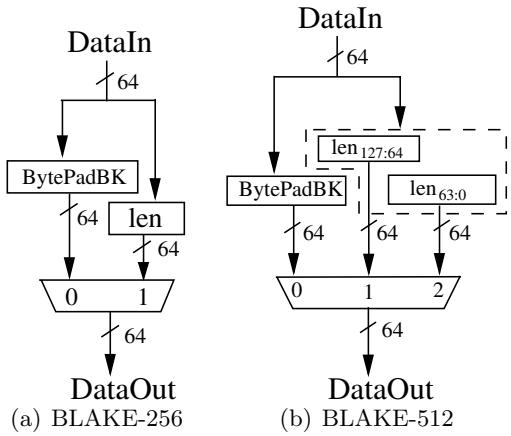
Figure 4.11: Boundary cases for BLAKE

Fig. 4.11 shows the possible three cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'.

For BLAKE -256:	For BLAKE -512:
Case(a) M $\leq$ 440,	Case(a) M $\leq$ 888
Case(b) 440 $<$ M $<$ 512,	Case(b) 888 $<$ M $<$ 1024
Case(c) M = 512.	Case(c) M = 1024

4.2.2 Block Diagram Description

Fig. 4.12 shows a simplified block diagram of top level of datapath for BLAKE. 2-input mux selects either the message passed through *BytePadBK* which pads the input if necesary or it selects the message length to pass to the hash core. Fig. 4.12(b) is for BLAKE-512, as in the 512-bit version the length field in padding rule is reprented in 128-bits, so it is split



(a) BLAKE-256 (b) BLAKE-512

Figure 4.12: Top level of Datapath of Padding unit: BLAKE.

into two and a 3 input mux selects the input to pass depending on the state of the message loading.

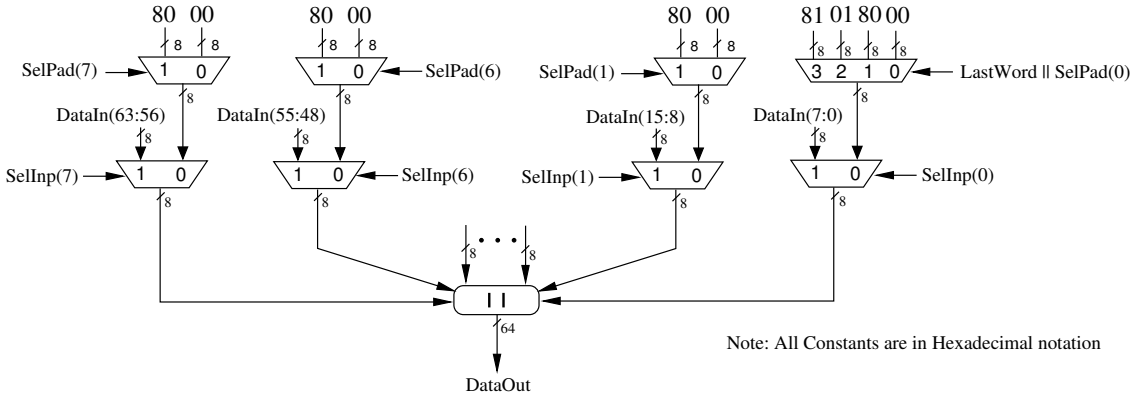


Figure 4.13: Block diagram of BytePadBK

Fig. 4.13 shows a simplified diagram of *BytePadBK*. The construction of the circuit is similar to the padding unit shown in fig. 4.2 with the exception of the last mux which takes care of the last byte in a word. The strings are precalculated depending on the different cases. *LastWord* is set by the controller when the last word of the last block is written to the hash core by the padding unit before it appends the message length. This facilitates in appending a trailing '1' to the message.

4.3 Grøstl

4.3.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

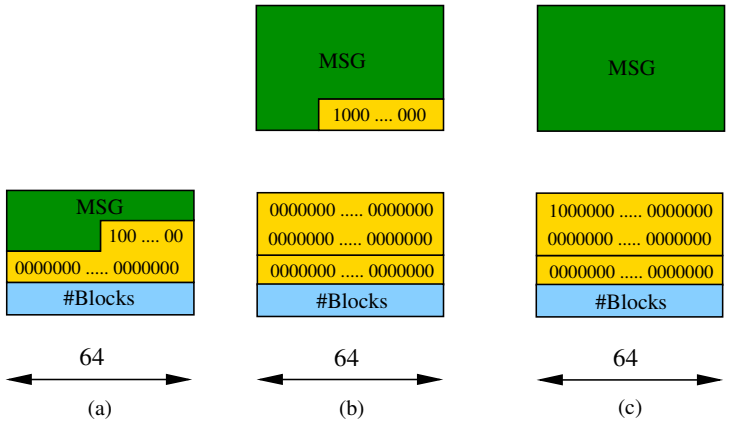


Figure 4.14: Boundary cases for Grøstl

Fig. 4.14 shows the possible three cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'

For Grøstl-256:

Case (a) M $\leq$ 440,

Case (b) 440 $<$ M $<$ 512,

Case (c) M = 512.

For Grøstl-512:

Case (a) M $\leq$ 952,

Case (b) 952 $<$ M $<$ 1024,

Case (c) M = 1024.

4.3.2 Block Diagram Description

Fig. 4.15 shows a simplified block diagram of top level of datapath of Padding unit for Grøstl. 2-input mux selects either the message passed through *BytePadMUL* which pads the input if necesary or selects the number of blocks in the message to pass to the core.

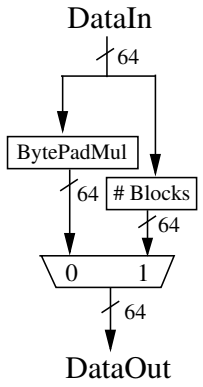


Figure 4.15: Top level of Datapath of Padding unit:Grøstl

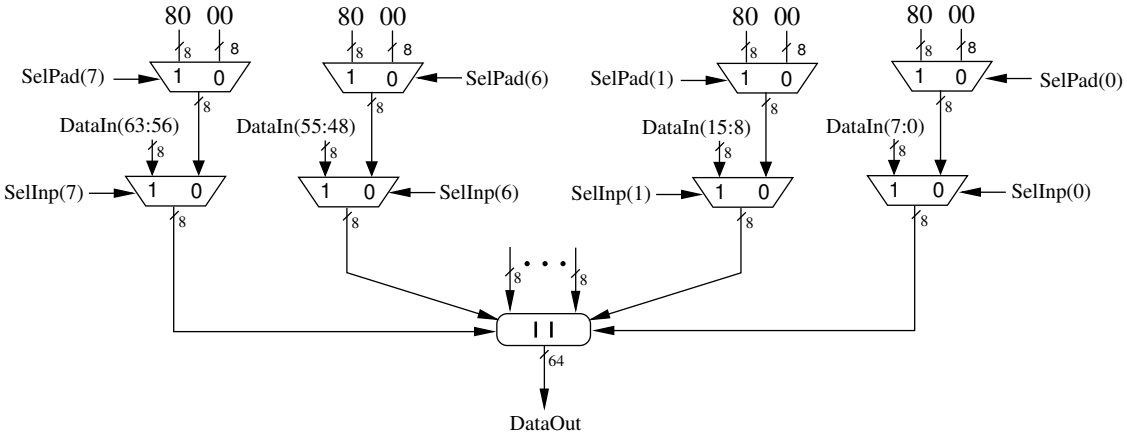


Figure 4.16: Block diagram of BytePadMUL

Fig. 4.16 shows a simplified diagram of *BytePadMUL*. The construction of the circuit is same as the padding unit shown in fig. 4.2. It either allows the input to pass or the padding string which could be either "80" or "00" depending on the select signals. This structure is used by JH and SHA-2 as well as they have similar padding rules.

4.4 JH

4.4.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

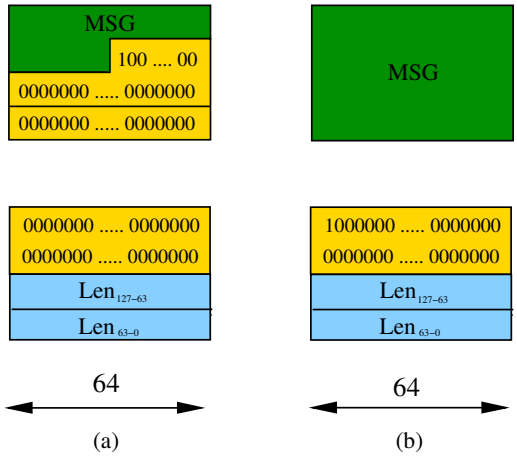


Figure 4.17: Boundary cases for JH

Fig. 4.17 shows the possible two cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'

For JH-256/512

Case (a) M $\leq$ 512,

Case (b) M = 512.

4.4.2 Block Diagram Description

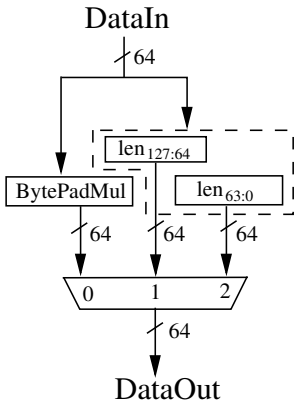


Figure 4.18: Top level of Datapath of Padding unit:JH

Fig. 4.18 shows a simplified block diagram of top level of datapath of Padding unit for JH. 3-input mux selects either the message passed through *BytePadMUL* which pads the input if necesary or it selects the higher 64-bits or the lower 64-bits of the message length to pass to the hash core.

4.5 Keccak

4.5.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

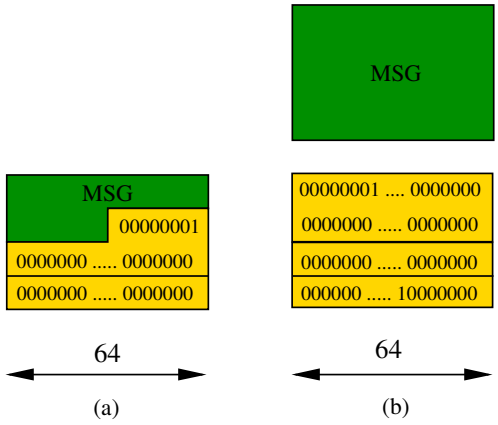


Figure 4.19: Boundary cases for Keccak

Fig. 4.19 shows the possible two cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'

For Keccak -256:	For Keccak -512:
Case (a) M $<$ 1088,	Case (a) M $<$ 576,
Case (b) M $=$ 1088.	Case (b) M $=$ 576.

4.5.2 Block Diagram Description

Fig. 4.20 shows top level of datapath of Padding unit for Keccak. Padding rule of Keccak has no counter so *BytePadKK* either pads the input if necessary or lets the input message pass to the hash core.

Fig. 4.21 shows a simplified diagram of *BytePadKK*. The construction of the circuit is similar to the padding unit shown in fig. 4.2 with the exception of the last mux which takes

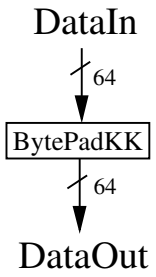


Figure 4.20: Top level of Datapath of Padding unit:Keccak

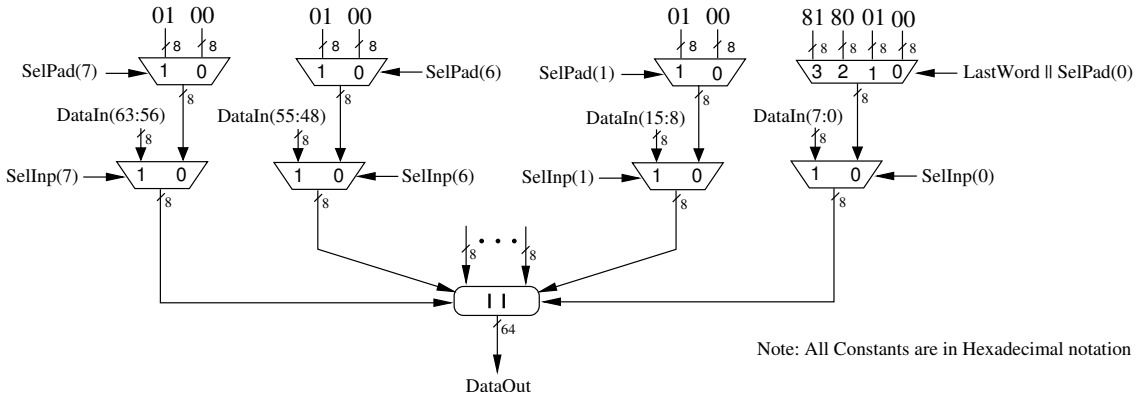


Figure 4.21: Block diagram of BytePadKK

care of the last byte in a word. The strings are precalculated depending on the different cases. *LastWord* is set by the controller when the last word of the last block is written to the hash core by the padding unit before it appends the message length. This facilitates in appending a trailing '1' to the message.

4.6 Skein

4.6.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

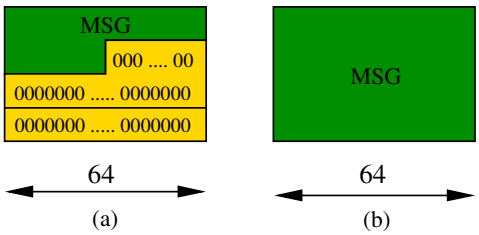


Figure 4.22: Boundary cases for Skein

Fig. 4.22 shows the possible two cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'

Skein -256/512

Case (a) M $<$ 512,

Case (b) M $=$ 512.

4.6.2 Block Diagram Description

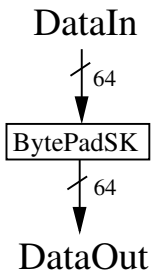


Figure 4.23: Top level of Datapath of Padding unit:Skein

Fig. 4.23 shows top level of datapath of Padding unit for Skein. Padding rule of Skein has no counter so *BytePadSK* either pads the input if necessary or lets the input message pass to the hash core.

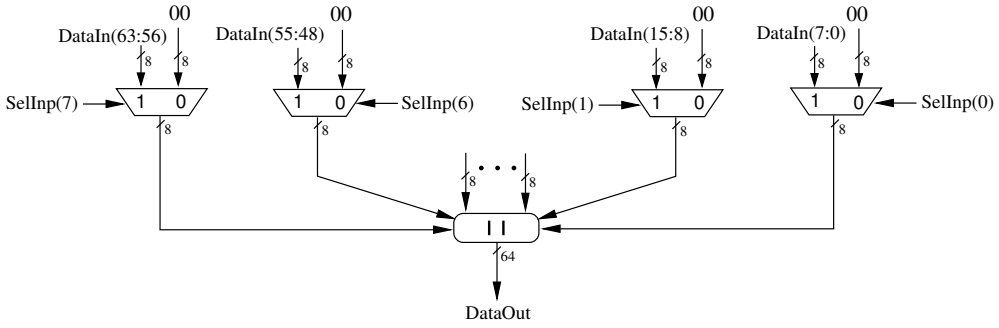


Figure 4.24: Block diagram of BytePadSK

Fig. 4.24 shows the diagram of *BytePadSK*. It is a very simple construction as skein just appends zeros if the message is not a multiple of block size. So only one level of mux is required.

4.7 SHA-2

4.7.1 Boundary cases

This section describes the possible boundary cases when the message size is a multiple of byte.

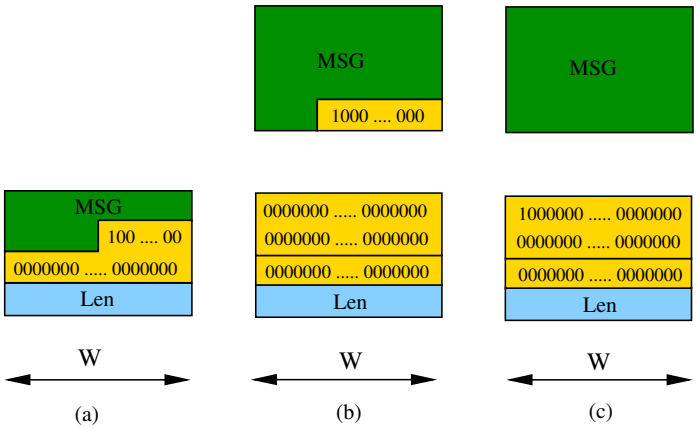


Figure 4.25: Boundary cases for SHA-2

Fig. 4.25 shows the possible three cases which can occur depending on the size of the last block. Suppose length of the last block is 'M'

For SHA-256:

Case (a) M $\leq$ 440,

Case (b) 440 $<$ M $<$ 512,

Case (c) M = 512.

For SHA-512:

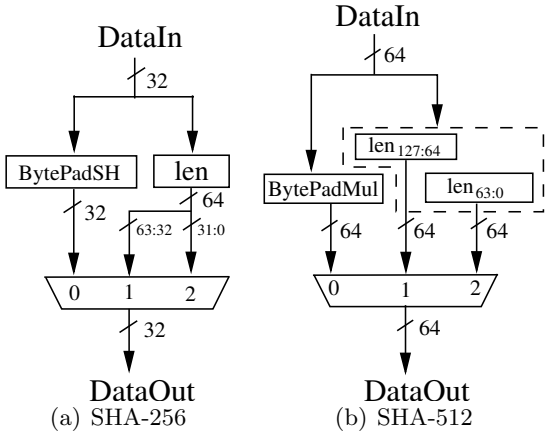
Case (a) M $\leq$ 888,

Case (b) 888 $<$ M $<$ 1024,

Case (c) M = 1024.

4.7.2 Block Diagram Description

Fig. 4.26 shows a simplified block diagram of top level of datapath for SHA2. 3-input mux selects either the message passed through *BytePadMUL* which pads the input if necesary or it selects the higher 32-bits of the message length or the lower 32-bits to pass to the hash



(a) SHA-256 (b) SHA-512

Figure 4.26: Top level of Datapath of Padding unit: SHA.

core. Fig. 4.26(b) is for SHA-512, as in the 512-bit version the length field in padding rule is reprented in 128-bits, so it is split into two and a 3 input mux selects the input to pass depending on the state of the message loading.

Chapter 5: Universal Padding Unit for ASICs

To implement and benchmark SHA-3 Round 3 finalists in ASICs and see how different optimization targets lead to different results, two groups George Mason University, Virginia USA (GMU) and Swiss Federal Institute of Technology Zurich (ETHZ) contributed one set of implementations each. Standard-cell based 65nm CMOS technology was used to implement all candidate algorithms and a reference implementation of SHA-2. 256-bit variants of all algorithms were implemented with Round 3 tweaks.

5.1 Core Interface

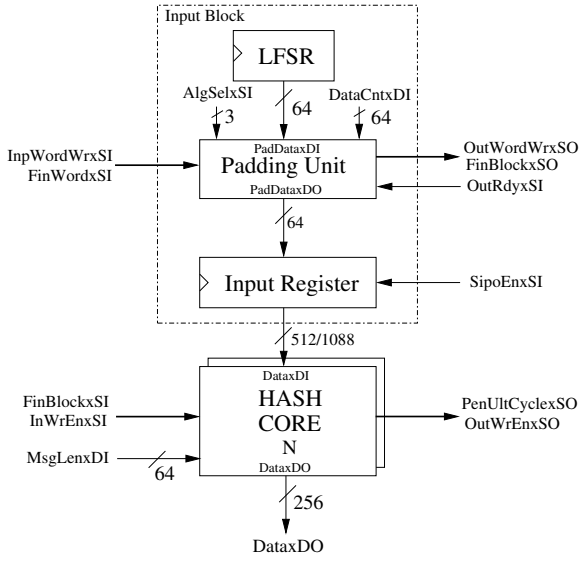


Figure 5.1: Simplified block diagram showing interface between Input block and Hash cores. Input bus width for keccak is 1088 and 512 for all others.

Due to pin limitations, an LFSR (using the primitive polynomial $x^{73} + x^{25} + 1$) was implemented which calculates 64-bits of output per clock cycle as a pseudo-random number

generator. The output of this LFSR becomes the input of the Padding Unit whose output is stored as a new input message block in an Input register which acts like a Serial In Parallel Out (SIPO). All cores get the message block in parallel as input as shown in Fig. 5.1. 4-bit *AlgSelxSI* selects the core to be active, and it also controls clock gating, thus only one core is active at any given time. Whenever a new message block is ready *InWrEnxSI* is set high to alert the core. *FinBlockxSI* signal is set high if the message block is the last block of the message. One clock cycle before the core can accept new data, it sets high PenUltCyclexSO signal. The core will assert *OutWrEnxSO* as soon as the 256-bit output is ready. The wrapper is expected to sample this output as soon as the *OutWrEnxSO* is active.

As area was limited on the chip having individual padding unit for each algorithm was not feasible. So a universal padding unit was developed which pads the input depending on *AlgSelxSI*. The padding unit should be small and fast enough not to slow any of the algorithm. Two padding units were developed, one assuming that the message length ends on the boundary of a byte and the other assumes that the message ends on the boundary of a word. Signals regarding the Padding unit which are independent of the core are discussed in the next section.

5.2 Universal Padding Unit-Byte version

This section provides details regarding the Interface signals and the design of the universal padding unit which assumes that the message ends at the boundary of a byte.

5.2.1 Interface

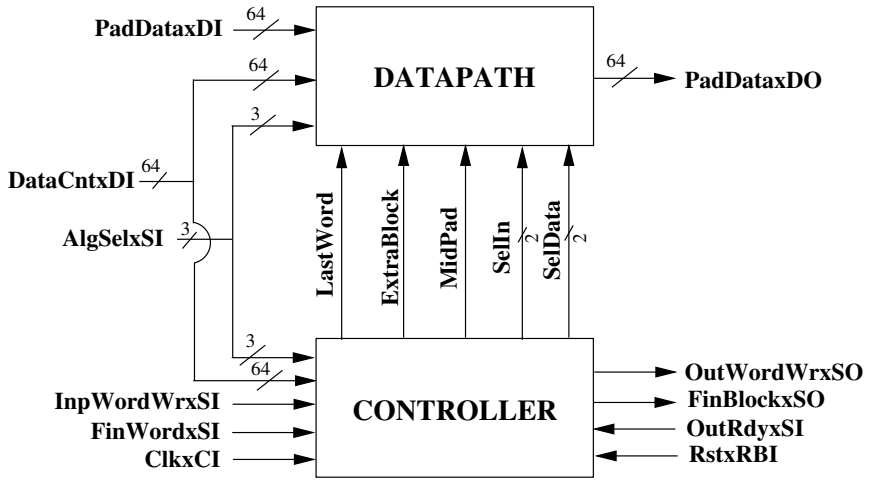


Figure 5.2: Byte version:Top level of Universal Padding Unit

Fig. 5.2 illustrates signals from the controller to the datapath. As can be seen from the figure datapath is completely combinational in structure with no clock input. *DataCntxSI* which is the output of a message length counter, is an input to the controller. It is used to calculate the time at which the message ends and thus gives us the information regarding start of padding. Table. 5.2.1 briefly describes function, width and names of the input, output and control signals. Padding unit only interfaces with the wrapper around it and has no direct communication with the hash core.

Table 5.1: Description of Interface Signals

Signal	Direction	Description
PadDataxDI	Input	64-Bit input data bus.
DataCntxDI	Input	64-Bit Message length (in bytes).
AlgSelxSI	Input	3-Bit Algorithm Select.
InpWordWrxSI	Input	Set high each time the next word is written to the Padding Unit.
FinWordxSI	Input	Set high when the last word is written to the Padding Unit.
ClkxCI	Input	Rising Edge Triggered global clock.
PadDataxDO	Input	64-Bit output data bus.
OutWordWrxSO	Output	Set high when the next word is written to the output.
FinBlockxSO	Output	Set high when the last word of the last block is written to the output.
OutRdyxSI	Input	Set High when the Padding unit can **start** writing the next word.
RstxRBI	Input	Asynchronous **Active low** reset.

5.2.2 Block diagram description

Fig. 5.3 shows the top level of the datapath. *BytePad* does the job of padding the message and takes *PadDataxDI* as an input. As described before in Table. 3.1 JH, BLAKE, and SHA-2 append message length at the end of the message. Depending on the algorithm, the length is represented by 128 or 64 bits. Grøstl appends total number of blocks in the message represented by 64 bits. *SelData* from the controller selects either the message or the other fields depending on *AlgSelxSI*. *DataCntxSI* is message length in bytes, so it is shifted to the left by 3 positions to get the message length in bits. Block size of Grøstl 256-bit variant is 512, so shifting the message length, which is a multiple of a byte by 6 positions to the right will give the total number of blocks in the message. Signal *ExtraBlock* is set high by the controller when the padding scheme results into an extra block which doesn't not contain any bits of message.

SelIn is a 2-bit signal generated by the controller to control the output of the LUTs. *MidPad* selects zeros which are to be padded in between. Lower three bits of *DataCntxDI*

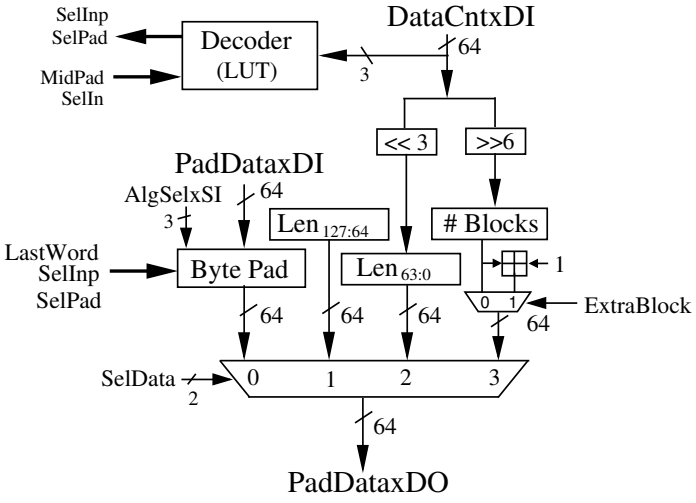


Figure 5.3: Byte version:Top level of datapath

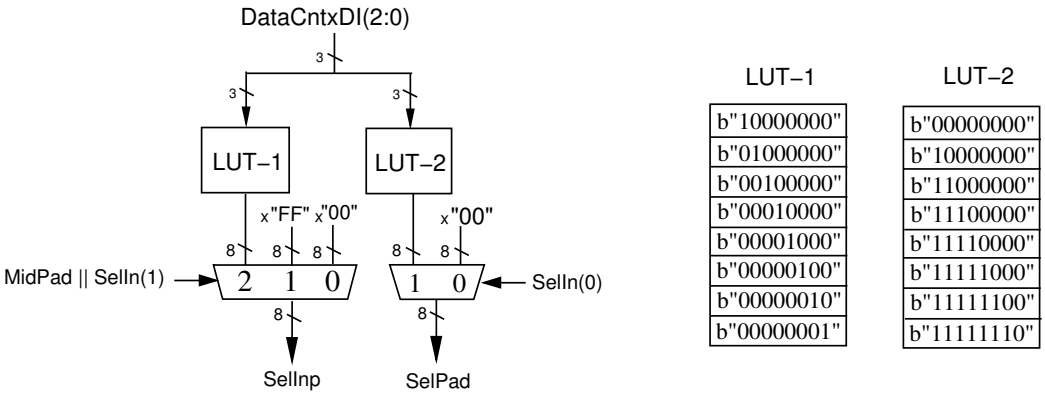


Figure 5.4: LUT based decoder to generate select signals for BytePad

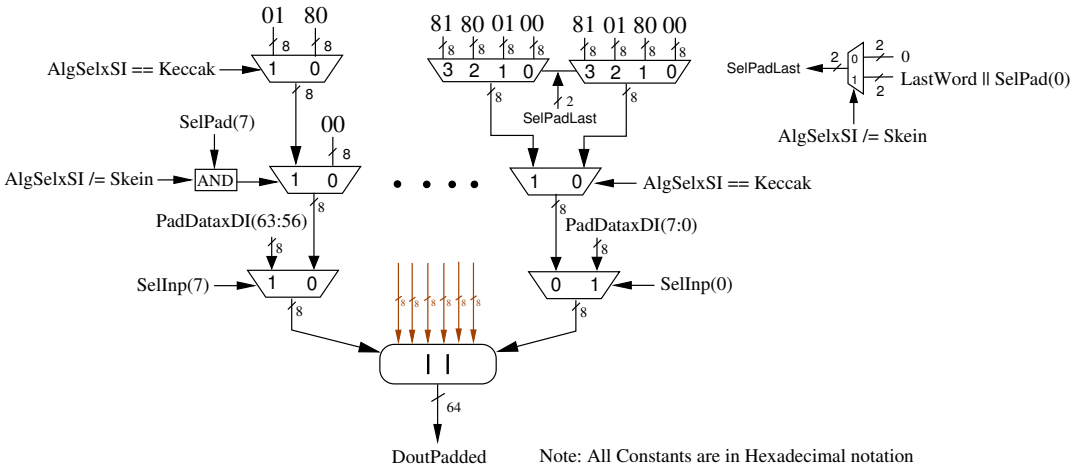


Figure 5.5: Block diagram for BytePad

are used as the address to the LUTs. LUT logic is similar as explained in section(blah).

As seen in Table. 3.1, all units from chapter Padding unit are very similar to each other. So all of them can be combined in one single universal unit and because of high level of comparability the circuit will not expensive. Starting string(x“80”) in all algorithms is the same except for Keccak. So a mux selects x“01” for keccak and x“80” for rest. All the algorithms add minimum number of 0s, so x“00” is common Mid padding string. Skein just append zeros for a message ending at the boundary of a byte, so if the algorithm is Skein, the rest of the values are ignored as shown in Fig. 5.5. The same structure is instantiated 7 times and the last byte is treated differently. Keccak and BLAKE append a trailing one at the end of the message, Thus 2 4x1 muxs are used at the last byte to accommodate the trailing ’1’. *LastWord* is set high by the controller when the last word of the last block is written to the hash core. *SelInp* and *SelPad* are generated using the LUT as shown in Fig. 5.4.

As seen in Table. 3.1, all units from chapter Padding unit are very similar to each other

5.3 Universal Padding Unit-Word version

This section provides details regarding the Interface signals and the design of the universal padding unit which assumes that the message ends at the boundary of a word(64-bits).

5.3.1 Interface

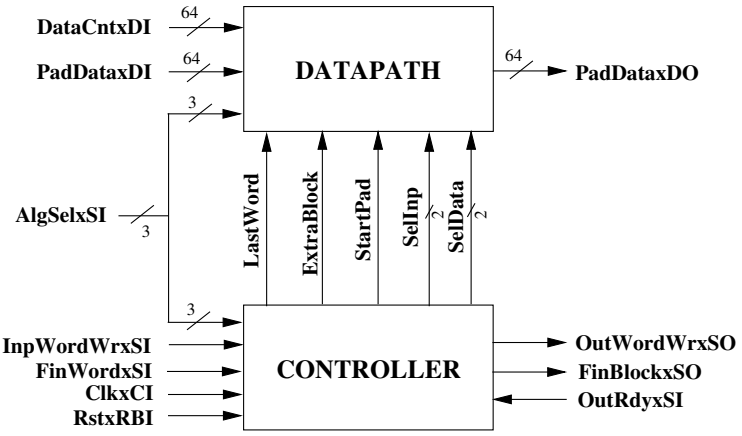


Figure 5.6: Word version:Top level of Universal Padding Unit

Fig. 5.6 illustrates signals from the controller to the datapath. The interface signals with the wrapper are similar to the one explained in section 5.2.1 . But control signals are different because now it is assumed that the message ends on the boundary of a word, thus one does not need to calculate at what position in the word the message ends. *FinWordxSI* indicates message end, which can be used as an indication to start padding.

As we assume that the message ends at the boundary of a word(64-bit), we don't need the LUT based decoder to generate the control signals. The controller generates *StartPad* and *SelInp* depending on where the message ends.

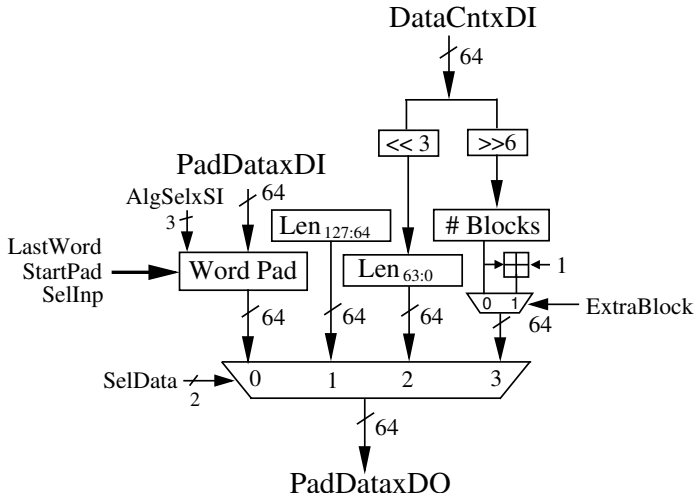


Figure 5.7: Word version:Top level of datapath

Top level of datapath is similar to the byte version, except few control signals. Word Pad unit is similar to the byte pad with the exception that the padding strings are of word lengths. Thus instead of many smaller multiplexers, a big 64x1 mux is used. The trailing '1' in Keccak and BLAKE is dealt with the same logic as explained in section 5.2.2, but now the strings are of word length.

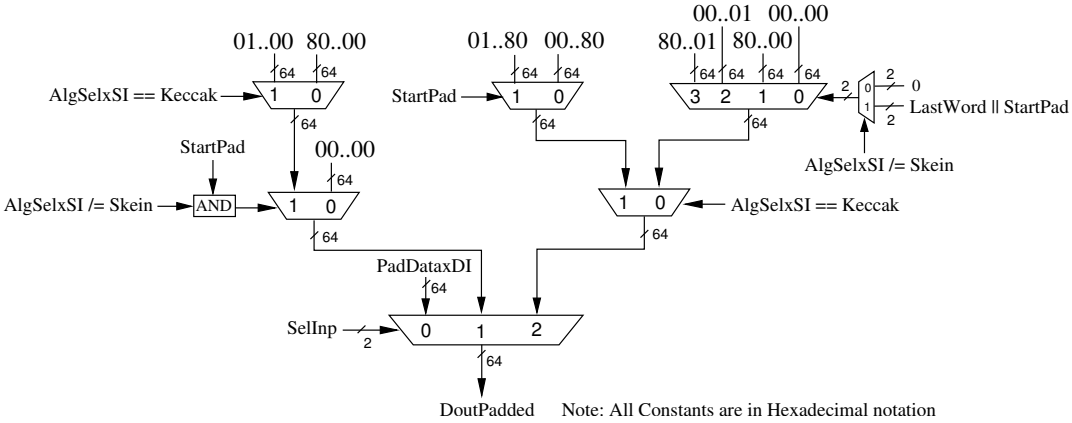


Figure 5.8: Block Diagram of word pad

Chapter 6: Hashed Message Authentication Code

6.1 Message Authentication Code

When two users are communicating with each other over a computer network which is unsecured, there needs to be a mechanism which provides message integrity and entity authentication. Message Authentication Code (MAC) is a secret key algorithm which provides message Integrity and authenticity [1].

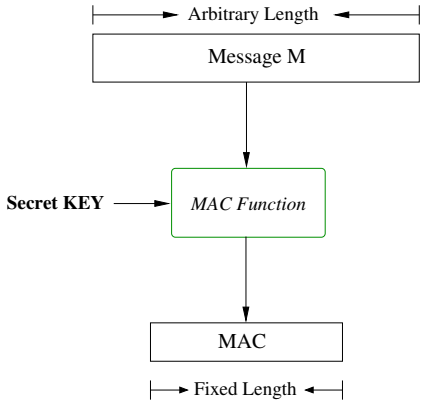


Figure 6.1: Simplified block diagram showing generation of MAC from arbitrary length message.

1. **Message Integrity** : It is a property which states that the message has not been modified by an unauthorized user after the transmission of the message by an authorized source.

2. **Message Source Authentication** : It is a property which states that message sent by one source is authentic and confirms the identity of the source.

As shown in Fig. 6.1, MAC function takes in arbitrary length message as an input to a compression function and the other input as a secret key shared between two users

and generates a fixed length output called a MAC or "authenticated tag". When user A transmits a message to user B over an insecure channel, he/she concatenates the calculated MAC value, generated using the shared secret key, with the message. On receiving the message, user B computes MAC using the same MAC function, with the key shared with user A as an input. If the MACs are equal then the source is corroborated and the message is assumed to be unchanged during transmission. MACs were generally implemented out of block ciphers, but using MACs with cryptographic hash functions is a better option, as hash functions are generally faster than block ciphers and the implementations are easily and freely available. Keying hash function for message authentication was proposed by [24] and is now a NIST standard [25].

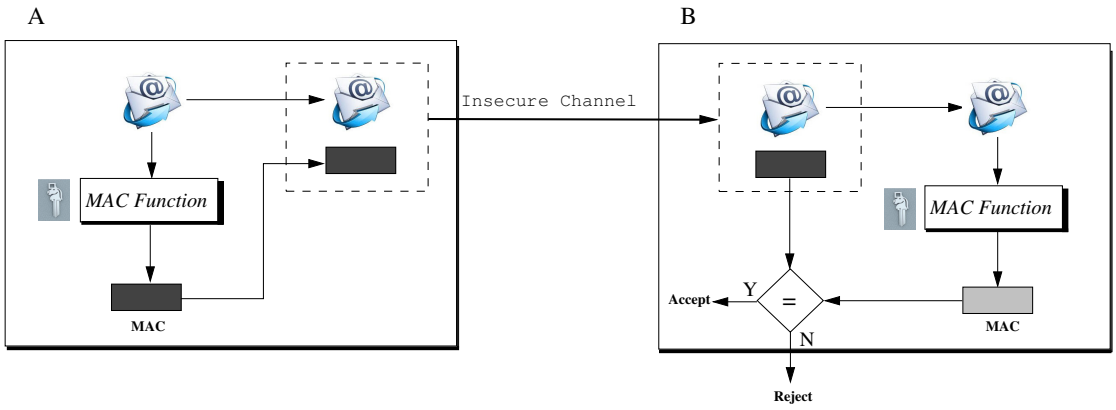


Figure 6.2: Communication between User A and User B over an unsecured channel.

6.2 HMAC

Hash based message authentication code (HMAC) is a shared secret key algorithm that uses hash functions for generating a MAC. Hash functions were not designed for entity authentication as they don't have secret key as part of the algorithm which are an indispensable aspect of message authentication. [24] came up with an algorithm which integrates secret key into the computation of hash function and called HMAC. The algorithm takes in two inputs, a key and a message which is transmitted and gives out a *MAC* or *authenticated tag.* The operation can be described by the equation:

HMAC(key, msg) = H((K$\oplus$opad)||H((K$\oplus$ipad)||msg))

The symbols used are listed below:

Key : Secret key shared between users communicating.

msg : Message which is to be transmitted and is used to calculate MAC.

H(x) : Hashing of data x using a hash function.

K : Key obtained after necessary preprocessing.

$\oplus$: XORing of two values.

|| : Concatenating two bit streams.

Step 1 Pre-Processing of key:

If size of the Key is equal to the block size of the hash function,

i.e. Key size = Block size, then K = Key.

If the size of the Key is less than the block size of the hash function, then the Key is padded with zeros at the right end so that the size of the key becomes equal to the block size.

i.e. Key < Block size; K = Key||"000...000".

If the size of the key is greater than the block size of the hash function, then the key is hashed first and then the hashed result is padded with zeros if necessary to get the final key.

K = H(Key)||"000...000"

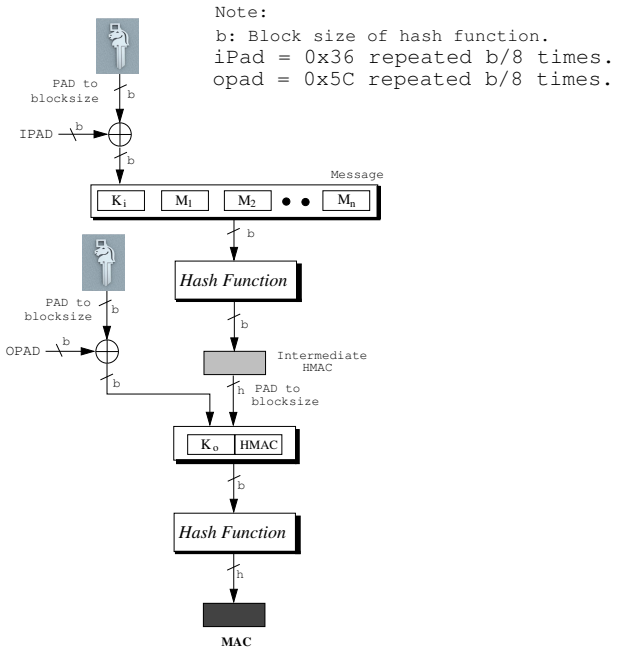


Figure 6.3: Details of HMAC algorithm

Step 2 XOR with ipad and HASH:

The obtained key after step 1 is XORed with **ipad(inner pad)** and the result is padded to the left of the message as shown in Fig. 6.3. Now the total message with the key is divided into blocks and processed using a hash function.

Step 3 XOR with opad and HASH:

The message digest obtained after the processing of the hash functions is the intermediate HMAC and is padded by the padding rule of a particular hash algorithm if necessary to make it equal to the block size. The key obtained from step 1 is XORed with **opad(outer pad)** and is appended to the left of the padded intermediate HMAC and is again passed through the hash function.

Step 4 Truncate MAC:

The final output from the hash function is called MAC and can be truncated from the right depending on the length of the desired MAC tag.

6.3 HMAC Wrapper

Assumptions and features:

The HMAC unit is designed to process all key sizes (Key≤ Block size and Key>Block size). The message size is assumed to be a multiple of a byte which is a limitation of the Hash core. Truncation to get a desired length MAC is assumed to be done by the user outside the HMAC unit. If the Key<Block size then the length has to be a multiple of the input word and if the Key>block size than the Key size can be in the multiple of a byte. The developed unit can reuse keys thus decreasing on computation time if using the same key and thus increasing throughput.

The HMAC algorithm [24] was designed with the idea in mind that the hash core can be treated as a black box and can be replaced by any approved Hash function by NIST. So a generic HMAC unit is designed to accommodate all Round 3 finalists and SHA-2. Due to this fact, a number of user defined constants are specified at the top level of VHDL code and can be modified by the user to select desired hash algorithm, output size and input word size.

6.3.1 Interface and Protocol

Input protocol is similar to what was used for the padding architecture described in section 2.1 before, as the HMAC unit is built as a wrapper on top of the SHA core. It supports signals *src_ready* and *dst_ready*, and outputs *src_read* and *dst_write*. In this interface FIFOs are connected at the input and output interface, but any module which can support the above mentioned signals can be used at the input-output interface.

Input protocol is similar to what was used for the padding architecture described in section(blah), the only difference being that now the first input is the key size and then the key itself. Which is followed by the message length and message itself.

But the input to the SHA core is modified to accommodate the HMAC algorithm. As seen from the Fig. 6.3 that after preprocessing the size of the key is equal to the block size of the hash algorithm thus in part 1 of the input the block size is sent to the core and then the

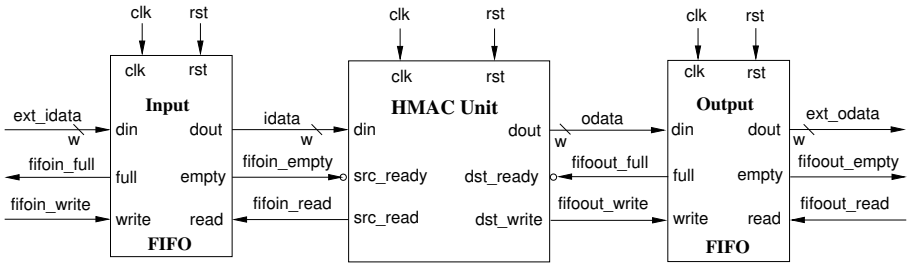


Figure 6.4: Interface for HMAC Unit

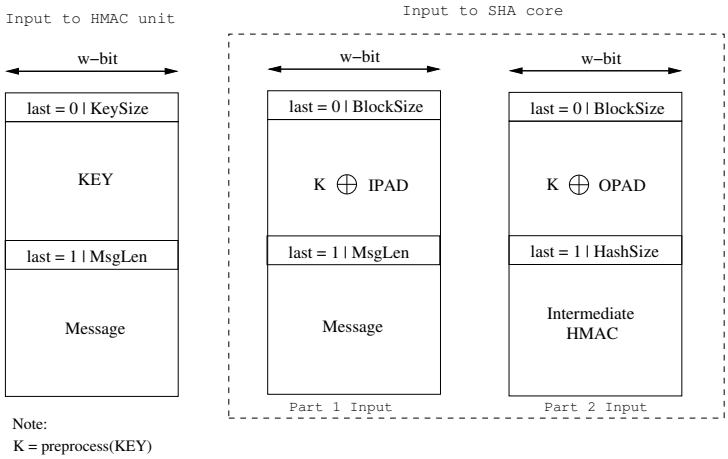


Figure 6.5: Protocol supported by HMAC wrapper

Key XORed with iPad. Then the message itself with its length. The part 2 is key XORed with opad and then the stored intermediate hash result. Width of the input depends on the width supported by the hash algorithm.

6.3.2 HMAC Datapath

The figure below is a simplified block diagram of the datapath.

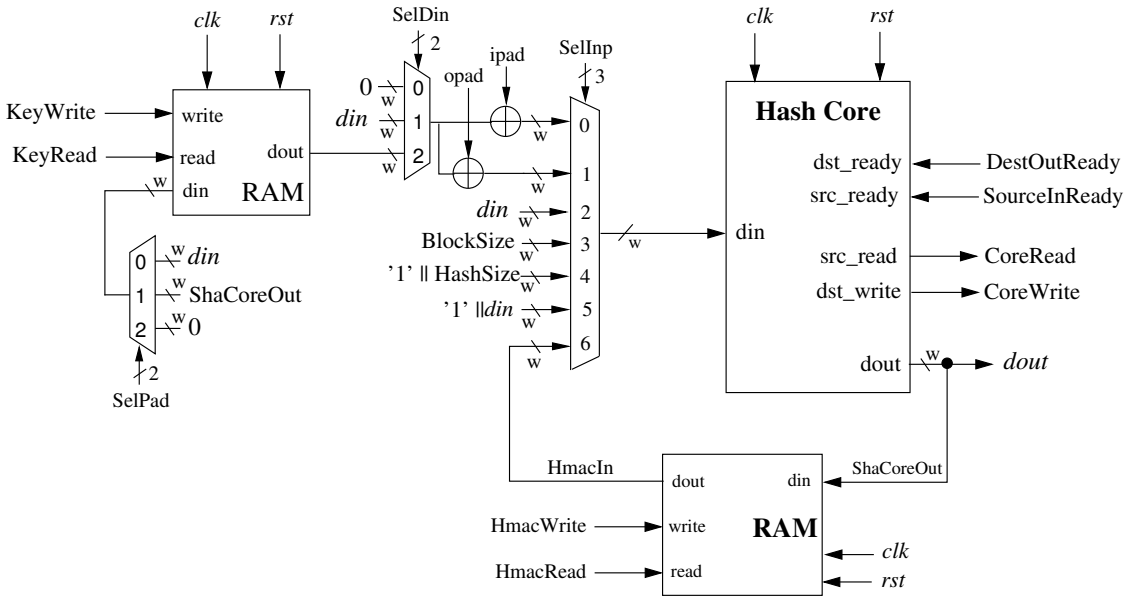


Figure 6.6: Datapath: HMAC wrapper

HMAC datapath consists of 2 RAMs and the SHA core. One RAM is used to store the key. The RAM has a mux at the input to select either an input key from the input source or the hashed key from the SHA core when the key is greater than the block size or zeros to pad when the key is smaller than the block size of the particular hash function. 'SelPad' is generated by the controller depending on the key size. The key is stored in the RAM so it can be reused for other messages, additionally the key is simultaneously stored in the RAM and also provided to the SHA core saving on time to store the key. As shown in Fig. 6.5 the SHA core receives the block size as an input and then the key itself. 'SelInp' generated from the controller selects the required input depending on the state of the computation. '1' is appended at the leftmost bit to indicate the end of message to the core. After the key is processed, 'SelInp' selects the original message to be passed to the SHA core by selecting 'Din'. 'HmacIn' is the intermediate HMAC stored in the output RAM. BlockSize and HashSize are constants and depend on the hash function implemented. The other RAM

is the output from the SHA core, it stores the intermediate HMAC and is read from after the XORing with opad state.

6.3.3 HMAC Top

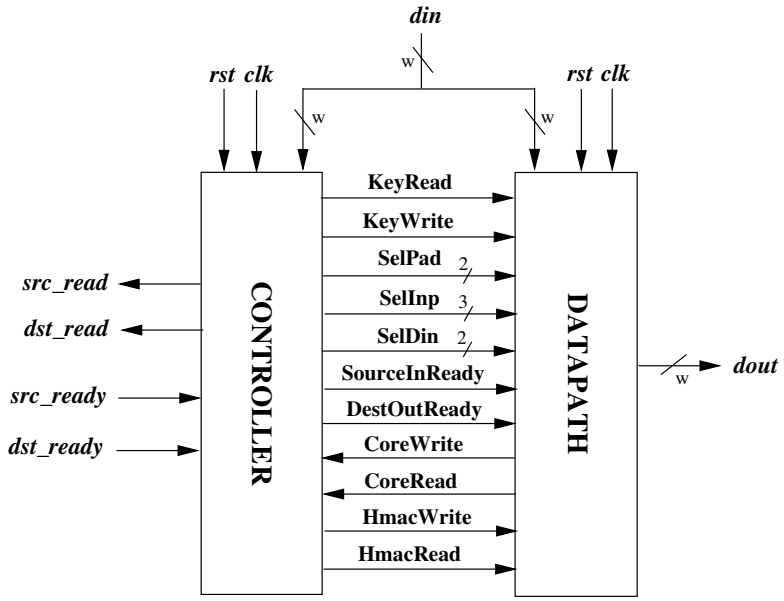


Figure 6.7: Controller-Datapath communication signals

The diagram shows the controller-datpath communication. The signals on the left of the diagram are the input pins on the chip. CONTROLLER provides all the necessary signals to the DATAPATH depending on the state. 'HmacWrite' and 'HmacRead' are the inputs to the RAM which stores the intermediate HMAC. All the control signals to the SHA core are provided by the controller.

Chapter 7: Results

7.1 Design Summary

7.1.1 Throughput

Throughput in terms of cryptographic algorithms can be defined as number of bits of a message processed within a specific time. The overall time taken by a hash function to process a message includes loading of messages from a source, initialization, computation, finalization and loading of hash digest to the destination. All results shown in this section are throughput for calculating long messages where all the other processes can be ignored and only computation times are taken into account. Taking into account these assumptions the formula for calculating throughtput is defined as follows:

$$Throughput = \frac{BlockSize}{T_{CLK} \cdot (\#cycles)}$$

Where BlockSize is the minimum number of bits required by the algortihm to start processing and is specific to the algorithm.T_{CLK} is the minimum clock period and #cycles is the number of clock cycles required to process one block of a message.

7.1.2 Area

The designs contains only basic elements and no dedicated resources such as Block RAMs, DSP units, multipliers, etc. Basic elements are specific to a given FPGA family. For Xilinx, the Basic elements are defined as the number of CLB slices. For the Altera Stratix family and more recent high performance families, Basic elements are defined as the number of Adaptive Look-up Tables (ALUTs).

Table 7.1: Througput equation for long messages with the I/O Data Bus width in bits, Throughput in Mbits/s.T_{CLK} denotes clock period in seconds

	256-bit variant		512-bit variant	
Algorithm	**I/O Bus Width**	**Throughput** [Mbit/s]	**I/O Bus Width**	**Throughput** [Mbit/s]
BLAKE	64	$512/(29 \cdot T_{CLK})$	64	$1024/(33 \cdot T_{CLK})$
Grøstl	64	$512/(21 \cdot T_{CLK})$	64	$1024/(29 \cdot T_{CLK})$
JH	64	$512/(43 \cdot T_{CLK})$	64	$512/(43 \cdot T_{CLK})$
Keccak	64	$1088/(24 \cdot T_{CLK})$	64	$576/(24 \cdot T_{CLK})$
Skein	64	$512/(73 \cdot T_{CLK})$	64	$512/(73 \cdot T_{CLK})$
SHA256	32	$512/(65 \cdot T_{CLK})$	64	$1024/(81 \cdot T_{CLK})$

7.1.3 Throughput/Area

All results are optimized for best thtoughput/area ratio. The formula for the Throughput to Area Ratio is:

$$Ratio = \frac{Throughput}{Area}$$

where Throuhgput and Area are the metrics described above.

Table. 7.1 lists all the formulas used for calculating the throughput for the algorithms. The results section is divided into three subsections. Section I shows results regarding padding unit in FPGAs, section II shows results for universal padding unit in ASICs and section III shows results for HMAC implementations.

7.2 Padding Unit-Results

Notation: Tp throughput, A area, Tp/A Throughput to Area Ratio, [%] relative change in the Throughput, Area, and Throughput to Area ratio as a result of adding padding unit to the hash unit.

Table 7.2 and 7.3 show results of all the algorithms implemented with the padding unit with 256 and 512-bit variants on 2 Xilinx and 2 Altera devices.

Table 7.2: The effect of the padding unit on the performance of 5 Round 3 SHA-3 finalists in 4 FPGA families. Virtex 5 & Virtex 6, Stratix III & Stratix IV.

Architecture	**Virtex 5**			**Virtex 6**			**Stratix III**			**Stratix IV**		
	Tp	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**
BLAKE-256												
No-Pad	2308	1771	1.30	2226	1257	1.77	2157	3553	0.61	2337	3543	0.66
Pad	2266	1860	1.22	2363	1391	1.70	2206	3660	0.60	2316	3680	0.63
Δ[%]	-1.83	5.03	**-6.53**	6.18	10.66	**-4.04**	2.25	3.01	**-0.74**	-0.90	3.87	**-4.59**
Grøstl-256												
No-Pad	6117	1795	3.41	7220	1870	3.86	6604	6460	1.02	6269	6421	0.98
Pad	6572	2020	3.25	7071	1884	3.75	6160	6466	0.95	6033	6415	0.94
Δ[%]	7.44	12.53	**-4.53**	-2.06	0.75	**-2.79**	-6.72	0.09	**-6.81**	-3.76	-0.09	**-3.67**
JH-256												
No-Pad	4955	982	5.05	5412	849	6.37	5276	3221	1.64	4759	3210	1.48
Pad	4543	1001	4.54	5086	918	5.54	5024	3383	1.49	4815	3415	1.41
Δ[%]	-8.32	1.93	**-10.06**	-6.02	8.13	**-13.09**	-4.77	5.03	**-9.33**	1.17	6.39	**-4.90**
Keccak-256												
No-Pad	13337	1369	9.74	11839	1086	10.90	15493	3531	4.39	16104	3471	4.64
Pad	12745	1375	9.27	12451	1147	10.86	14624	4060	3.60	15167	3734	4.06
Δ[%]	-4.44	0.44	**-4.86**	5.16	5.62	**-0.43**	-5.61	14.98	**-17.91**	-5.82	7.58	**-12.45**
Skein-256												
No-Pad	3023	1218	2.48	3373	1005	3.36	2475	3943	0.63	2592	3936	0.66
Pad	3127	1245	2.51	2957	1026	2.88	2495	3960	0.63	2647	3970	0.67
Δ[%]	3.43	2.22	**1.19**	-12.33	2.09	**-14.13**	0.77	0.43	**0.34**	2.10	0.86	**1.23**
SHA-256												
No-Pad	1400	396	3.54	1633	239	6.83	1656	959	1.73	1797	959	1.87
Pad	1559	427	3.65	1645	309	5.33	1673	1122	1.49	1739	1122	1.55
Δ[%]	11.36	7.83	**3.11**	0.70	29.29	**-21.96**	1.03	17	**-13.87**	0.66	14.04	**-12.15**

Both tables show the effect of adding padding unit to the algorithms to the over all throughput/area ratio.The designs were optmized for throughput/area ratio, so the percentage effect is marked in bold. Fig. 7.1 and 7.2 plots the change in throughput/area ratio for Altera devices and Fig. 7.3 and 7.4 for Xilinx.

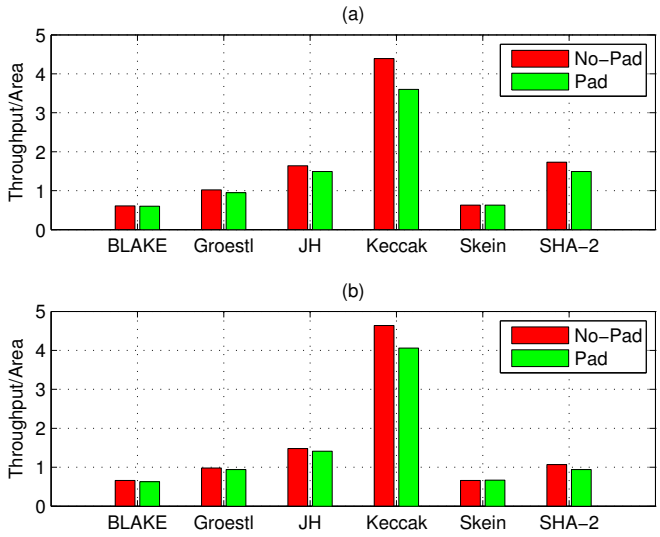


Figure 7.1: Change in throughput/area ratio after adding of padding unit in Altera devices for 256-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Stratix III and (b) is Stratix IV

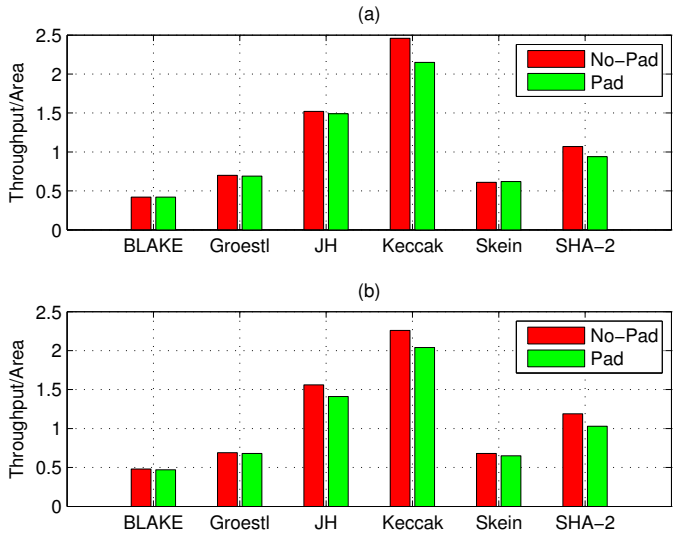


Figure 7.2: Change in throughput/area ratio after adding of padding unit in Altera devices for 512-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Stratix III and (b) is Stratix IV

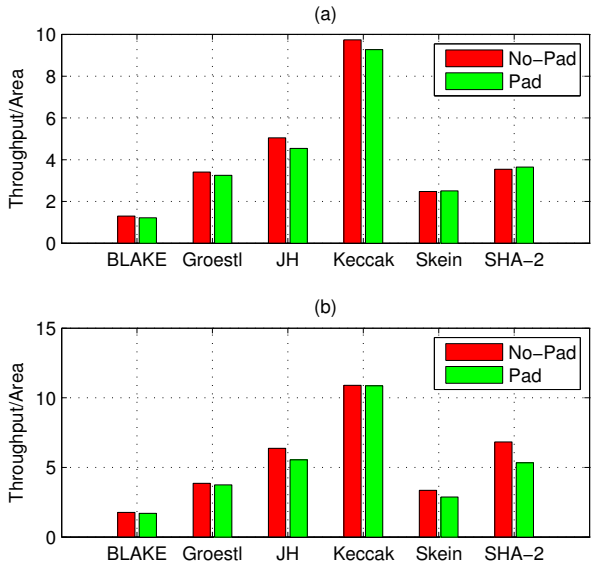


Figure 7.3: Change in throughput/area ratio after adding of padding unit in Xilinx devices for 256-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Virtex 5 and (b) is Virtex 6

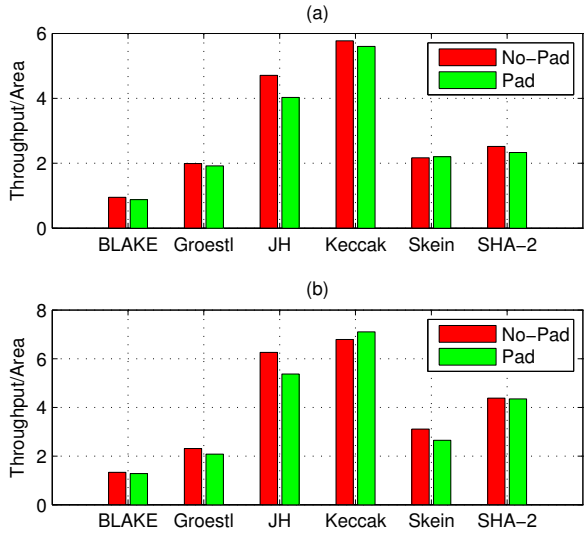


Figure 7.4: Change in throughput/area ratio after adding of padding unit in Xilinx devices for 512-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Virtex 5 and (b) is Virtex 6

Table 7.3: The effect of the padding unit on the performance of 512-bit variant of 5 Round 3 SHA-3 finalists in 4 FPGA families.Virtex 5 & Virtex 6 , Stratix III & Stratix IV.

Architecture	**Virtex 5**			**Virtex 6**			**Stratix III**			**Stratix IV**		
	Tp	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**
BLAKE-512												
No-Pad	3264	3435	0.95	3478	2610	1.33	2928	6977	0.42	3318	6971	0.48
Pad	3156	3569	0.88	3333	2608	1.28	3003	7115	0.42	3320	7114	0.47
Δ[%]	-3.29	3.90	**-6.92**	-4.18	-0.08	**-4.10**	2.55	1.98	**0.56**	0.06	2.05	**-1.96**
Grøstl-512												
No-Pad	7686	3853	1.99	8375	3630	2.31	8669	12450	0.70	8504	12368	0.69
Pad	7462	3895	1.92	6843	3285	2.08	8638	12570	0.69	8507	12562	0.68
Δ[%]	-2.92	1.09	**-3.96**	-18.29	-9.50	**-9.71**	-0.37	0.96	**-1.32**	0.02	1.57	**-1.52**
JH-512												
No-Pad	4882	1037	4.71	5825	931	6.26	5011	3288	1.52	5139	3294	1.56
Pad	4531	1125	4.03	4834	901	5.37	5024	3383	1.49	4815	3415	1.41
Δ[%]	-7.18	8.49	**-14.44**	-17.01	-3.22	**-14.25**	0.25	2.89	**-2.56**	-6.31	3.67	**-9.63**
Keccak-512												
No-Pad	7612	1320	5.77	7208	1061	6.79	8526	3471	2.46	7825	3467	2.26
Pad	7179	1283	5.60	7465	1052	7.10	8029	3734	2.15	7607	3723	2.04
Δ[%]	-5.68	-2.80	**-2.96**	3.56	-0.85	**4.45**	-5.82	7.58	**-12.45**	-2.79	7.38	**-9.47**
Skein-512												
No-Pad	3084	1418	2.17	3462	1114	3.11	2438	4006	0.61	2736	4015	0.68
Pad	2972	1348	2.20	3141	1186	2.65	2493	4035	0.62	2597	4026	0.65
Δ[%]	-3.64	-4.94	**1.36**	-9.28	6.46	**-14.79**	2.24	0.72	**1.51**	-5.07	0.27	**-5.33**
SHA-512												
No-Pad	2012	798	2.52	2421	553	4.38	2128	1995	1.07	2377	1996	1.19
Pad	2026	870	2.33	2398	551	4.35	2142	2275	0.94	2390	2311	1.03
Δ[%]	0.70	9.02	**-7.54**	-0.95	-0.36	**-0.68**	0.66	14.04	**-12.15**	0.55	15.78	**-13.45**

7.3 Universal Padding Unit-Results

Table. 7.4 shows the area and maximum clock frequnecy of the universal padding unit in ASICs. The maximum clock frequnecy is far greater than the fastest core from both groups. The maximum area overhead is less than 6% for the SHA-3 finalists. Table. 7.5 lists all the implementations results, i.e. Area and Maximum clock frequency and the percentage overhead in area due to the universal padding unit.

Table 7.4: Area and maximum clock frequency results for both version of universal padding unit on ASIC

	Version	**Area [kGE]**	**Max.Clock[MHz]**
Padding Unit	Byte	2.13	1428.5
	Word	2.08	1428.5

Table 7.5: Area and maximum clock frequency results for implemented cores 256-bit variant on ASIC

Algorithm	**Group**	**Area** [kGE]	**Max.Clock**[MHz]	**Overhead**[%]
BLAKE	GMU	43.02	252.40	4.95
	ETHZ	39.96	201.41	5.33
Grøstl	GMU	160.28	459.14	1.33
	ETHZ	69.39	273.15	3.07
JH	GMU	54.35	602.77	3.92
	ETHZ	46.79	330.58	4.55
Keccak	GMU	80.65	599.16	2.64
	ETHZ	46.31	485.40	4.60
Skein	GMU	71.90	179.86	2.96
	ETHZ	71.87	349.41	2.96
SHA-2	GMU	25.14	537.63	8.47
	ETHZ	24.30	294.55	8.77

Fig. 7.5 and7.6 plots the are overhead of the universal padding unit in ASICs and FPGAs respectively. Fig. 7.5(a) is the plot for GMU implementation and (b) is for ETHZ implementations of the core. Fig. 7.6 is the plot for GMU cores implemented on FPGAs showing the area overhead due to the universal padding unit. Fig. 7.6(a) is the plot of Virtex 5 and (b) for Stratix 3. Table. 7.6 and 7.7 show area and maximum clock frequency results of universal padding unit for Xilinx and Altera FPGA devices.

Table 7.6: Area and maximum clock frequency results for both version of universal padding unit on Xilinx FPGAs

		Virtex 5		**Virtex 6**	
	Version	**Area** [CLB slices]	**Max.Clock** [MHz]	**Area** [CLB slices]	**Max.Clock** [MHz]
Padding Unit	Byte	89	597.37	122	481.93
	Word	89	580.72	93	581.73

The Table. 7.8 shows results for cores from GMU implemented on FPGAs to show area overhead due to universal padding unit for virtex 5 and Stratix III family.

Table 7.7: Area and maximum clock frequency results for both version of universal padding unit on Altera FPGAs

		Stratix III		Stratix IV	
	Version	**Area** [ALUTs]	**Max.Clock** [MHz]	**Area** [ALUTs]	**Max.Clock** [MHz]
Padding Unit	Byte	340	526.59	340	558.04
	Word	293	538.79	293	558.79

Table 7.8: Area and maximum clock frequency results for both version of universal padding unit on Altera FPGAs

	Virtex 5			Stratix III		
Algorithm	**Area** [ALUTs]	**Max.Clock** [MHz]	**Overhead** [%]	**Area** [ALUTs]	**Max.Clock** [MHz]	**Overhead** [%]
BLAKE	1515	122.249	8.05	3444	124.63	3.54
Grøstl	1719	207.211	7.10	7491	258.2	1.63
JH	898	406.174	13.59	3370	425.3	3.62
Keccak	1235	291.545	9.88	4115	302.57	2.96
Skein	1291	112.208	9.45	3939	91.55	3.10
SHA-2	625	230.415	19.52	1026	213.49	11.89

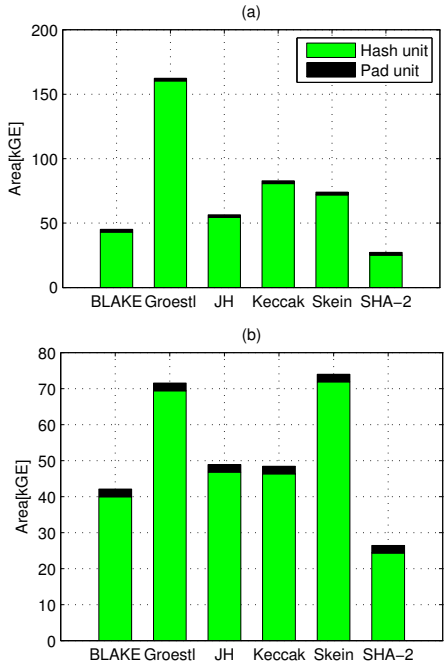


Figure 7.5: Area overhead due to addition of universal padding unit byte version for GMU and ETHZ implementations in ASIC.

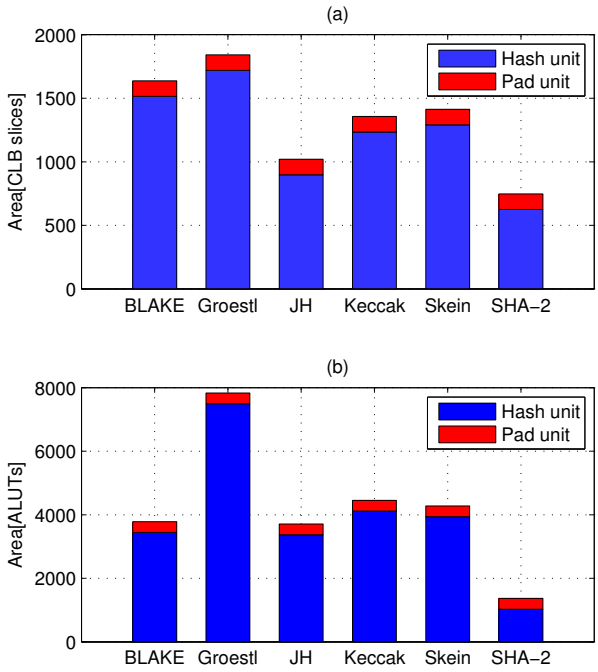


Figure 7.6: Area overhead due to addition of universal padding unit byte version for GMU implementations in FPGA.

7.4 HMAC Wrapper-Results

Notation: Tp throughput, A area, Tp/A Throughput to Area Ratio, [%] relative change in the Throughput, Area, and Throughput to Area ratio as a result of adding padding unit to the hash unit.

Table 7.9 and 7.10 show results of all the algorithms implemented with the HMAC wrapper with 256 and 512-bit variants on 2 Xilinx and 2 Altera devices.

Table 7.9: The effect of the HMAC WRAPPER on the performance of 256-bit variant of 5 Round 3 SHA-3 finalists in 4 FPGA families.Virtex 5 & Virtex 6 , Stratix III & Stratix IV.

Architecture	**Virtex 5**			**Virtex 6**			**Stratix III**			**Stratix IV**		
	Tp	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**
BLAKE-256												
Hash Core	2266	1860	1.22	2363	1391	1.70	2206	3660	0.60	2316	3680	0.63
HMAC	1615	1417	1.14	2318	1405	1.65	2158	4240	0.51	2324	4272	0.54
Δ[%]	-28.71	-23.82	**-6.56**	-1.89	1.01	**-2.94**	-2.17	15.85	**-15.55**	0.36	16.09	**-13.55**
Grøstl-256												
Hash Core	6572	2020	3.25	7071	1884	3.75	6160	6466	0.95	6033	6415	0.94
HMAC	6285	1893	3.32	6326	1669	3.79	6349	7595	0.84	5787	7477	0.77
Δ[%]	-4.37	-6.29	**2.15**	-10.54	-11.41	**1.07**	3.08	17.46	**-12.24**	-4.08	16.55	**-17.70**
JH-256												
Hash Core	4543	1001	4.54	5086	918	5.54	5024	3383	1.49	4815	3415	1.41
HMAC	3531	1226	2.88	3713	990	3.75	3519	3744	0.94	3483	3745	0.93
Δ[%]	-22.28	22.48	**-36.56**	-27.01	7.84	**-32.31**	-29.95	10.67	**-36.70**	-22.22	9.66	**-34.04**
Keccak-256												
Hash Core	12745	1375	9.27	12451	1147	10.86	14624	4060	3.60	15167	3734	4.06
HMAC	12230	1470	8.32	12885	1251	10.30	14632	3483	4.20	149353	3746	3.99
Δ[%]	-4.04	6.91	**-10.25**	3.49	9.07	**-5.16**	0.06	-14.21	**16.63**	-1.53	0.32	**-1.84**
Skein-256												
Hash Core	3127	1245	2.51	2957	1026	2.88	2495	3960	0.63	2647	3970	0.67
HMAC	3093	1452	2.13	3570	1182	3.02	2494	4535	0.55	2639	4535	0.58
Δ[%]	-1.09	16.63	**-15.14**	20.72	15.20	**4.86**	-0.01	14.52	**-12.69**	-0.28	14.23	**-12.70**
SHA-256												
Hash Core	1559	427	3.65	1645	309	5.33	1673	1122	1.49	1739	1122	1.55
HMAC	1670	497	3.36	1614	339	4.76	1694	1180	1.44	1752	1187	1.48
Δ[%]	7.11	16.39	**-7.95**	-1.91	9.71	**-10.69**	1.28	5.17	**-3.62**	0.75	6.74	**-4.77**

Both tables show the effect of adding HMAC wrapper to the algorithms to the over all throughput/area ratio. The designs were optmized for throughput/area ratio, so the percentage effect is marked in bold. Fig. 7.7 and 7.8 plots the change in throughput/area ratio for Altera devices and Fig. 7.9 and 7.10 for Xilinx.

Table 7.10: The effect of the HMAC WRAPPER on the performance of 512-bit variant of 5 Round 3 SHA-3 finalists in 4 FPGA families.Virtex 5 & Virtex 6 , Stratix III & Stratix IV.

Architecture	**Virtex 5**			**Virtex 6**			**Stratix III**			**Stratix IV**		
	Tp	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**	**Tp**	**A**	**Tp/A**
BLAKE-512												
Hash Core	3156	3569	0.88	3333	2608	1.28	3003	7115	0.42	3320	7114	0.47
HMAC	3232	3941	0.82	3707	2737	1.30	2999	7956	0.37	3359	8036	0.41
Δ[%]	2.41	10.42	**-6.82**	11.22	4.95	**1.56**	-0.12	11.82	**-10.24**	1.18	12.96	**-11.06**
Grøstl-512												
Hash Core	7462	3895	1.92	6843	3285	2.08	8638	12570	0.69	8507	12562	0.68
HMAC	7583	3970	1.91	8526	3723	2.29	8306	14675	0.56	8015	14625	0.54
Δ[%]	1.62	1.93	**-0.52**	24.59	13.33	**10.10**	-3.84	16.75	**-17.97**	-5.79	16.42	**-19.41**
JH-512												
Hash Core	4531	1125	2.70	4119	1012	4.07	5024	3383	1.49	4815	3415	1.41
HMAC	3216	1191	4.71	5825	931	6.26	3486	4054	0.86	3515	4054	0.87
Δ[%]	-29.03	5.87	**-33**	-14.79	12.32	**-24.21**	-19.09	3.16	**-21.82**	-16.33	3.44	**-18.97**
Keccak-512												
Hash Core	7179	1283	5.60	7465	1052	7.10	8029	3734	2.15	7607	3723	2.04
HMAC	6799	1368	4.97	7367	1117	6.60	7547	3274	2.31	7330	3293	2.23
Δ[%]	-5.29	6.63	**-11.2**	6.18	-7.11	**4.45**	-6.01	-12.32	**7.21**	-3.64	-11.55	**9.12**
Skein-512												
Hash Core	2972	1348	2.20	3141	1186	2.65	2493	4035	0.62	2597	4026	0.65
HMAC	2903	1528	1.90	3584	1294	2.77	2485	4645	0.54	2644	4638	0.57
Δ[%]	-2.31	13.35	**-13.64**	14.12	9.11	**4.53**	-0.32	15.12	**-13.71**	1.80	15.20	**-12.31**
SHA-512												
Hash Core	2026	870	2.33	2398	551	4.35	2142	2275	0.94	2390	2311	1.03
HMAC	2232	938	2.38	2431	683	3.56	2696	3142	0.85	2994	3132	0.95
Δ[%]	10.19	7.82	**2.15**	1.40	23.96	**-18.16**	25.86	38.11	**-8.72**	25.28	35.53	**-7.18**

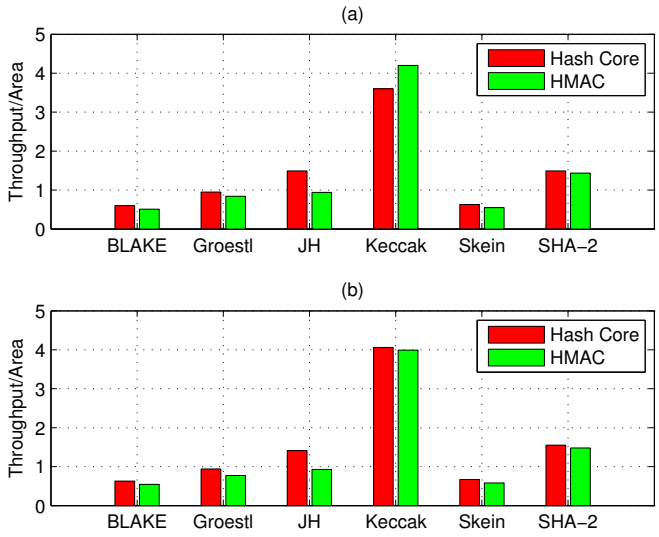


Figure 7.7: Change in throughput/area ratio after adding of HMAC wrapper in Altera devices for 256-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Stratix III and (b) is Stratix IV

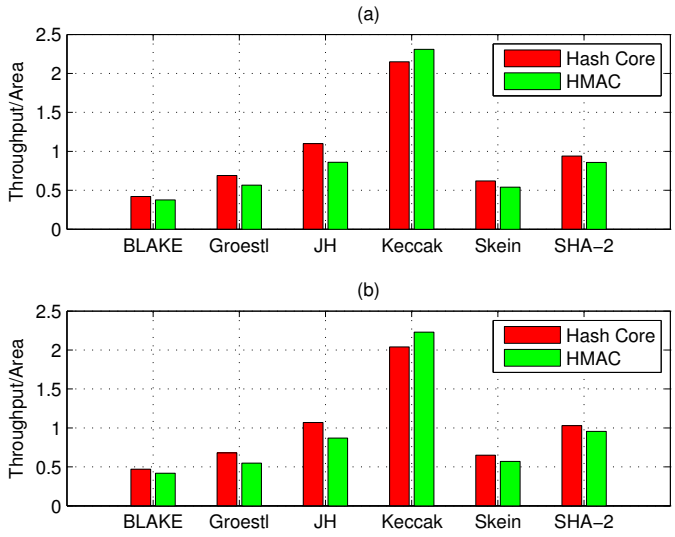


Figure 7.8: Change in throughput/area ratio after adding of HMAC wrapper in Altera devices for 512-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Stratix III and (b) is Stratix IV

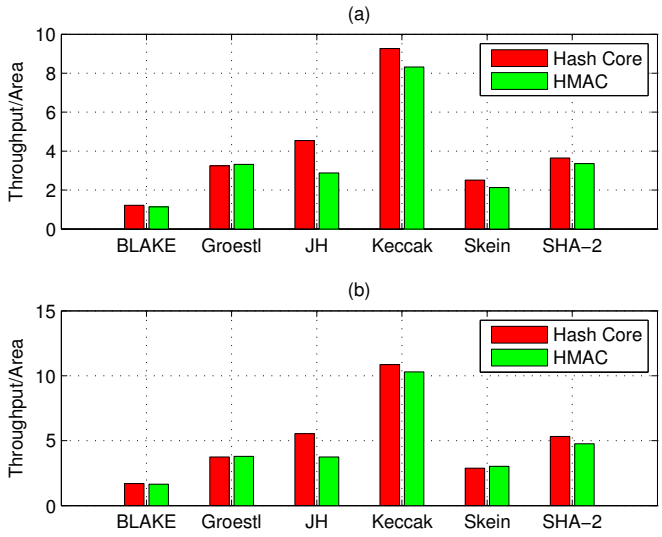


Figure 7.9: Change in throughput/area ratio after adding of HMAC wrapper in Xilinx devices for 256-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Virtex 5 and (b) is Virtex 6

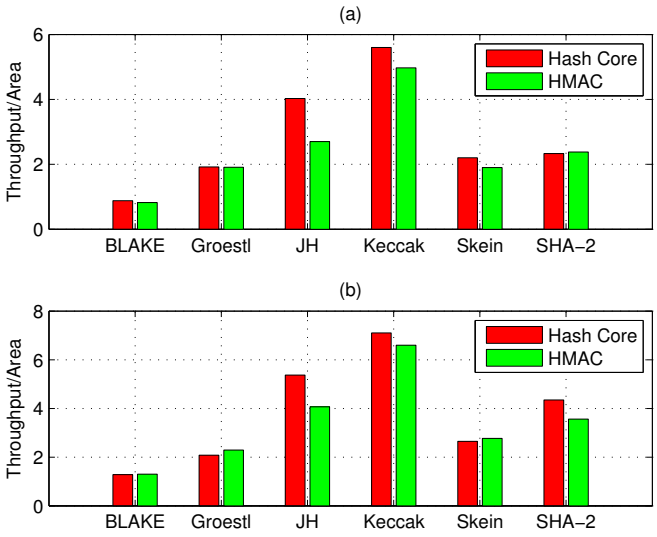


Figure 7.10: Change in throughput/area ratio after adding of HMAC wrapper in Xilinx devices for 512-bit variant for all SHA-3 finalists and SHA-2. (a) is graph for Virtex 5 and (b) is Virtex 6

Chapter 8: Conclusion and Future work

8.1 Conclusion

In this thesis, individual padding units for all 5 Round 3 SHA-3 finalists and SHA-2 for reference were designed, implemented, tested, and analyzed. Padding rule accommodating message length is the most challenging because of the counter and additional controller logic. The worst affected in terms of the throughput/area ratio in 256-bit variants was implementation of JH. This is because JH has a 128-bit adder and the core itself is very fast and small. Skein is the least affected as it has a very simple padding scheme with no counter and appends just zeros to the message. 512-bit variants follow similar trend as 256-bit where JH is the most affected and Skein do not show any decrease in throughput/area except for Virtex 6. BLAKE also show very small decrease in throughput/area ratio, just because BLAKE is one of the biggest of the 5 algorithms and also has less throughput.

The aim to build a universal padding unit for an ASIC chip was to have a single small unit which could accommodates all the padding rules and not increase the critical path. Universal padding unit can run at a maximum clock frequency of 1.428 GHz which is faster than all the algorithms.The area is 2.13 kGE for the byte version which results in only around maximum of 6% area overhead, which is for in ETHZ implementation of BLAKE.

8.2 Future work

Implementing all algorithms supporting padding in hardware on FPGA boards and doing experimental testing to see if the testing results are the same as what obtained after Post place and routing.

Bibliography

Bibliography

[1] A. J. Menezes, P. C. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press Inc., 1997.

[2] I. Damgard, "A design principle for hash functions," in *CRYPTO*, 1989, pp. 416–427.

[3] ——, "Collision free hash functions and public key signature schemes," in *EUROCRYPT*, 1987, pp. 203–216.

[4] R. Merkle, "SECRECY, AUTHENTICATION, AND PUBLIC KEY SYSTEMS," Ph.D. dissertation, Stanford University, Stanford, California, June 1979.

[5] M. Nathan, *War at Sea: A Naval History of World War II*. New York: Oxford University Press, 1995.

[6] N. Mridul, "Characterizing Padding rules of MD Hash functions preserving Collision Security," in *Proceeding of Information Security and Privacy*, vol. 5594/2009. Springer, 2009, pp. 171–184.

[7] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering:Design Principles and Practical Applications*. Indianapolis: Wiley Publishing, Inc., 2010.

[8] *Secure Hash Standard (SHS)*, National Institute of Standards and Technology (NIST), FIPS Publication 180-2, Aug 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[9] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "A hardware wrapper for the SHA-3 hash algorithms," Cryptology ePrint Archive, Report 2010/124, 2010, http://eprint.iacr.org/.

[10] B. Jungk and J. Apfelbeck, "Area-efficeint FPGA implementations of the SHA-3 finalists," in *International Conference on ReConfigurable Computing and FPGAs*. IEEE: ReConfig'11, DEC 2011, accepted, to be published.

[11] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O'Neill, and W. P. Marnane, "FPGA implementations of the round two SHA-3 candidates," Second SHA-3 Candidate Conference, Tech. Rep., 2010.

[12] M.-Y. Wang, H. C.-T. Su, Chih-Pin, and C.-W. Wu, "An HMAC processor with integrated SHA-1 and MD5 algorithms," in *Asia and South Pacific Design Automation Conference*, 2004.

[13] E. Khan, W. El-Kharashi, and F. Gebali, "Design and performance analysis of a unified, reconfigurable HMAC-hash unit," *IEEE Transactions on Circuits and Systems*, vol. 54, no. 12, pp. 2683–2695, DEC 2007.

[14] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the FULL SHA-1," *CRYPTO*, pp. 1–20, 2005, also available at http://www.springerlink.com/content/26vljj3xhc28ux5m.

[15] R. Rivest, "The MD5 message-digest algorithm," MIT Laboratory for Computer Science and RSA Data Security Inc., RFC 1321, Apr 1992.

[16] M. Juliato and C. Gebotys, "FPGA implementation of an HMAC processor based on the SHA-2 family of hash functions," University of Waterloo, Tech. Rep., 2011.

[17] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGA," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer Berlin / Heidelberg, 2010, pp. 264–278.

[18] E. Homsirikamol, "Fair and comprehensive comparison of hardware performance of SHA-3 Round 2 Candidates using FPGAs," Master's thesis, George Mason University, 2010.

[19] E. Homsirikamol, M. Rogawski, and K. Gaj, "Throughput vs. area trade-offs architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs," in *Workshop on Cryptographic Hardware and Embedded Systems CHES 2011*, ser. LNCS, B. Preneel and T. Takagi, Eds. Springer Berlin / Heidelberg, Sep 2011.

[20] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa – Automated Tool for Hardware EvaluatioN: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," in *20th International Conference on Field Programmable Logic and Applications - FPL 2010*. IEEE, 2010, pp. 414–421, http://cryptography.gmu.edu/athena.

[21] "GMU SHA-3 source codes," ONLINE, 2011, http://cryptography.gmu.edu/athena/.

[22] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs," Cryptology ePrint Archive, Report 2010/445, 2010, http://eprint.iacr.org/.

[23] S. Drimer, T. Güneysu, and C. Paar, "DSPs, BRAMs and a pinch of logic: Extended recipes for AES on FPGAs," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 3, no. 1, pp. 1–27, 2010.

[24] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," Network Working Group, RFC 2104, Feb 1997.

[25] *The Keyed-Hash Message Authentication Code (HMAC)*, National Institute of Standards and Technology (NIST), FIPS Publication 198, Mar 2002, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

Curriculum Vitae

Ambarish vyas was born on July 22nd, 1987 in Jodhpur, Rajasthan India. He received his Bachelor of Engineering Degree in Electronics and Telecommunications from University of Pune, India in 2009. He started working towards his Master of Science degree in Computer Engineering in University of Maryland from August 2009 and later transferred to George Mason University in August 2010. During the course of his studies, he was involved in teaching various undergraduate courses as a Teaching Assistant. Also a part of Cryptographic Engineering Research Group (CERG), his focus was on efficient implementation of cryptographic algorithms, embedded systems and physical VLSI design.