TOOLS AND EXPERIMENTAL SETUP FOR EFFICIENT HARDWARE BENCHMARKING OF CANDIDATES IN CRYPTOGRAPHIC CONTESTS

by

Farnoud Farahmand

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:

_____	Dr. Kris Gaj, Thesis Director
_____	Dr. Jens-Peter Kaps, Committee Member
_____	Dr. Avesta Sasan, Committee Member
_____	Dr. Monson Hayes, Chairman, Department of Electrical and Computer Engineering
_____	Dr. Kenneth S. Ball, Dean, The Volgenau School of Engineering

Date: _____ Fall Semester 2016
George Mason University
Fairfax, VA

Tools and Experimental Setup for Efficient Hardware Benchmarking of Candidates in Cryptographic Contests

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

By

Farnoud Farahmand
Bachelor of Science
Islamic Azad University, Karaj Branch, 2013

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

Fall Semester 2016
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents, Sedigheh Ghadakpour and Forood Farahmand. The lessons they have taught me and their continued support have allowed me to overcome many challenges during my graduate studies. Their unconditional love and encouragement have without any doubt led to my success. I am truly thankful for having them in my life.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Dr. Kris Gaj for continuous support of my Master's study and related research, and for his patience, motivation, and immense knowledge. His guidance helped me at all times during research and writing of this thesis.

In addition to my advisor, I would like to thank the rest of my thesis committee: Dr. Jens-Peter Kaps and Dr. Avesta Sasan, for their insightful comments and encouragement, but also for their hard questions which incentivized me to widen my research from various perspectives.

I would also like to thank my uncle, Mohammad Ghadakpour who supported me at the beginning of this step.

Last but not least, I would like to thank my family, my parents and my sister, for supporting me spiritually throughout writing this thesis and life in general.

Table of Contents

List of Tables

List of Figures

Abstract

TOOLS AND EXPERIMENTAL SETUP FOR EFFICIENT HARDWARE BENCHMARKING OF CANDIDATES IN CRYPTOGRAPHIC CONTESTS

Farnoud Farahmand

George Mason University, 2016

Thesis Director: Dr. Kris Gaj

Hardware benchmarking of candidates competing in cryptographic contests, such as SHA-3 and CAESAR, is very important for ranking of their suitability for standardization. A huge amount of time is necessary to design the datapath and controller and convert them to the hardware description language (HDL) code, due to an increasing number of candidates. The other difficulty is to develop a testbench in HDL for verification purposes. High-Level Synthesis (HLS), based on the newly developed Xilinx Vivado HLS tool, offers a potential solution to the aforementioned problems. Therefore, in the first part of this thesis we investigate the following hypothesis: Ranking of candidate algorithms in cryptographic contests in terms of their performance in modern FPGAs & All-Programmable SoCs will remain the same independently whether the HDL implementations are developed manually or generated automatically using HLS tools. In order to verify a potential validity of this approach, 4 Round 2 SHA-3 candidates are implemented using Vivado HLS and compared with existing RTL implementation. Our results indicate that the ranking of the evaluated candidates, in terms of four major performance metrics, frequency, throughput, area, and throughput to area ratio, has remained unchanged for all tested candidates.

In addition, one of the most essential performance metrics is the throughput, which highly depends on the algorithm, hardware implementation architecture, coding style, and options of tools. The maximum throughput is calculated based on the maximum clock frequency supported by each algorithm. A common way of determining the maximum clock frequency is static timing analysis provided by the CAD toolsets, such as Xilinx ISE, Xilinx Vivado, and Altera Quartus Prime. Finding actual maximum clock frequency utilizing static timing analysis is not a trivial task, especially in the Xilinx Vivado environment. It is extremely time consuming and tedious. As a result, in the second part of this thesis, we describe Minerva. Minerva is an automated hardware benchmarking tool which finds maximum frequency based on static timing analysis. It can be configured to target either Throughput or Throughput/Area as optimization criteria and to search through specific number of optimization strategies. The tool determines the best requested clock frequency, leading to the maximum value of the optimization target. We evaluated 20 Round 2 CAESAR candidates in terms of frequency and frequency to area ratio. Minerva frequency search is compared to binary search and results demonstrated up to 37% improvement in terms of throughput to area ratio and up to 24% in terms of throughput.

In the third part of the thesis, we have developed a universal testbed, which is capable of measuring the maximum clock frequency experimentally, using a prototyping board. We are targeting cryptographic hardware cores, such as implementations of SHA-3 candidates. Our testbed is designed using a Zynq platform and takes advantage of software/hardware co-design and Advanced eXtensible Interface (AXI). We measured the maximum clock frequency and the execution time of 12 Round 2 SHA-3 candidates experimentally on ZedBoard and compared the results with the frequencies reported by Xilinx Vivado. Our results indicate that depending on the characteristics of each algorithm, we may achieve either much higher or the same experimental frequency than the results reported by the tools using static timing analysis.

Chapter 1: Introduction

Cryptographic contests have emerged as a commonly accepted way of developing cryptographic standards. This process has appeared to work very well in case of Advanced Encryption Standard (AES), developed in the period 1997-2001 [1], and Secure Hash Algorithm 3 (SHA-3), developed in the period 2007-2012 [2]. At the same time, the observed increase in the number of algorithms qualified to the first round of the respective contests (51 in case of SHA-3 and 57 for CAESAR) inevitably brings the question of the sustainability of the current approach and its applicability to the development of future cryptographic standards. The number of candidates submitted to the first round of SHA-3 (51) has easily exceeded the number of submissions to any previous contest, confirming the aforementioned trend. Similarly, the numbers of candidates qualified to the second rounds of the respective competitions have increased from 5 in case of AES, through 14 for SHA-3, to 29 in case of CAESAR.

In November 2007, NIST announced SHA-3 public competition to develop a new cryptographic hash function standard. About four years have been spent on evaluating candidates submitted to this contest in terms of security, software and hardware efficiency, simplicity, and flexibility. Fifty one candidates were qualified to the first round of the contest. Their number was reduced first to fourteen and then to five candidates, in the second and the third round of the competition, respectively. The majority of candidates qualified to Round 2 were judged to have adequate security, and thus their performance in software and hardware became a decisive factor. Traditional hardware benchmarking consists of 3 steps: 1- Translation of specification to a hardware description language (HDL) like VHDL and Verilog, 2- Writing a testbench based on test vectors generated by reference software implementation for functional verification, 3- Generating post-place and route results using FPGA tools and performing timing verification of the obtained netlist. All of these steps

take significant amount of time. The objective of the second section of this thesis is to use High Level Synthesis for hardware development to decrease the time that is required for this task. HLS lets us take advantage of high-level programing language, such as C, C++ to generate synthesizable HDL. HLS implementation of five finalists has been reported in [3]. In this project, we implemented another four Round 2 candidates using Vivado HLS. As a result, we can rank all of these candidates in term of performance in modern FPGAs and compare the results with manual RTL implementation.

Throughput, area, and throughput to area ratio were the most important metrics used for hardware evaluation. Throughput of a hash function is the number of message bits for which a hash value (digest) can be computed per unit of time. In hardware, the maximum throughput depends on the maximum clock frequency supported by each algorithm, the block size, and the number of clock cycles required to process a block. Maximum clock frequency that can be achieved by a hardware implementation can be estimated or measured at different stages of the design process. The main stages are synthesis, placing and routing (P&R), and actual implementation on the board. The post-synthesis and post place & route results are determined by the FPGA tools using static timing analysis. Hardware evaluation of 14 round 2 SHA-3 candidates, based on post-placing and routing results, is reported in [4]. There are two difficulties associated with static timing analysis of digital systems designed and modeled using hardware description languages, and implemented using FPGAs:

1. The latest version of CAD tool provided by Xilinx (Vivado), does not have the capability to report the maximum frequency achievable for the corresponding design. Essentially, the user requests a target frequency and the tool reports a pass or fail, for its attempt to achieve this goal.

2. While there are 25 optimization strategies (i.e., sets of preselected option values) predefined in the tool, applying them sequentially, especially using Graphical User Interface, is extremely tedious and time consuming.

To overcome the aforementioned difficulties and facilitate hardware benchmarking of

algorithms by static timing analysis method, we introduce Minerva in the third section. Minerva is an automated and comprehensive hardware benchmarking tool. Minerva employs a unique searching algorithm, which is customized for frequency search using CAD toolsets, in addition to support for other standard search techniques. It can incorporate an arbitrary number of predefined or user defined strategies to achieve the highest possible frequency for each design. Moreover, it takes advantage of multithreading and multi core execution to reduce the run time significantly.

Accordingly, in this section, we will provide the Minerva frequency search results in terms of Throughput, Area and Throughput to Area ratio for 20 Round 2 CAESAR candidates. Then, we compare them with the results generated using binary search. Additionally, the run time of both methods (Minerva and binary search) are reported for 11 candidates.

In the fourth section, we demonstrate that the interface of Hash Core proposed in [4] can be easily combined with the de-facto industry standard, AMBA AXI [5], in order to achieve the practical, industry-grade designs for hardware accelerators implemented using reconfigurable logic of All Programmable Systems on Chip (SoC), such as Zynq. We also investigate the communication overhead introduced by the transfer of data between these hardware accelerators and the microprocessor core, for various sizes of data inputs. Secondly, we explore how the maximum clock frequency reported by Xilinx Vivado and the actual frequency measured experimentally compare to each other. Thus, we are verifying the accuracy of the worst-case values of the maximum clock frequency, reported by static timing analysis.

Our expectation is that the experimental clock frequency should be greater or equal than the worst-case value returned by the tools for all investigated algorithms. Finally, we also compare the ratios of the maximum experimental clock frequency to the maximum frequency returned by the tools. The straightforward expectation could be that these ratios should be approximately the same for all algorithms, as any given instance of the Zynq device is likely to operate with a frequency higher by certain specific percentage than the worst case instance of the same integrated circuit. However, as shown in this paper, this

naive expectation appears to be very far from the true behavior of a particular Zynq device observed for various investigated algorithms.

It should be stressed that we do not advocate replacing the well-established benchmarking methodology based on post-place and route results with the comparison of experimental values, as these values are more accurate only for a specific instance of the Zynq device, and cannot be generalized to millions of similar devices operating in the field, affected to a different extent by variations in the fabrication process.

Chapter 2: HLS-Based Implementation of Round-2 SHA-3 Candidates

2.1 High Level Synthesis

HLS allows designers to use a high-level programming language, such as C, C++, or Java, to generate synthesizable HDL. [6] concluded that until recently, this approach had been impractical due to the inadequate quality and high cost of HLS tools. This situation has changed when Xilinx acquired AutoESL Design Technologies Inc. in 2011, and incorporated its HLS tool, AutoPilot, into its latest toolchain, Vivado, in 2012 [7]. The availability of the industrial-quality, low-cost tool has allowed the HLS-based design approach to become more realistic. Previous studies have demonstrated that HLS can reduce the development time, while maintaining good performance as compared to software [8–10]. It has also been shown that a design using HLS-based approach can compete against a hand-written Register Transfer Level (RTL) code [11–15], at least in selected domains. [8] and [9] examined Autopilot HLS tool in three different application domains: MiBench benchmark, computer vision (stereo matching) and cryptography (AES, TDES and SHA). Their results demonstrated that 4x to over 126x speedup was achieved in comparison with software implementations with a five-fold reduction in design effort vs. manual design in HDL. However, RTL results were still better than those obtained using HLS.

In [3], the authors implemented 5 SHA-3 finalists using both high level synthesis and RTL. Throughput, area, frequency and throughput over area results for two different FPGA families were provided for all of these algorithms. This study has demonstrated that despite a noticeable performance penalty, caused by the use of high-level synthesis tools vs. manual design, the ranking of the evaluated candidates, in terms of four major performance metrics,

frequency, throughput, area, and throughput to area ratio, has remained unchanged for Altera Stratix IV FPGAs.

2.2 Research Approach and Ideas

2.2.1 Traditional and New Hardware Benchmarking Flows

A traditional, RTL-based hardware benchmarking flow, shown in Fig. 2.1, typically starts with the translation of an informal specification to a hardware description language (HDL) code, e.g., the RTL code written in VHDL or Verilog. This translation involves developing a block diagram of the Datapath and an Algorithmic State Machine (ASM) chart of the Controller. After manual design is completed, functional verification is performed based on test vectors generated using a reference software implementation. Then, the post-place & route results and the final netlist are generated. In order to make this step as efficient as possible, we can apply two approaches shown in Fig. 2.1. For Virtex 6, Spartan 6, and older FPGA families, Xilinx ISE (an older-generation Xilinx CAD tool) and ATHENa [16] could be used for the comprehensive optimization of FPGA tool options. Similarly, for Xilinx 7 Series FPGAs and Zynq, Xilinx Vivado (a new-generation Xilinx CAD tool) and its 25 default optimization strategies could be adopted. Finally, the obtained netlist undergoes a timing verification as the last check.

The proposed HLS-based development and benchmarking approach is shown in Fig. 2.2. In this methodology, a designer starts from a reference software implementation, which is typically provided by the algorithm designers in C. This implementation is then manually transformed into an HLS-ready C code, by the addition of HLS tool directives, provided as pragmas, and necessary optimizations that make the C code more suitable for parallelization and resource reuse. Once these modifications are completed, the code is verified in software for the correct functionality. Afterward, the HLS-ready C code is processed by the HLS tool to generate an RTL HDL code. This code is further processed using the tools and optimization strategies described above for the RTL flow. If the obtained results are worse

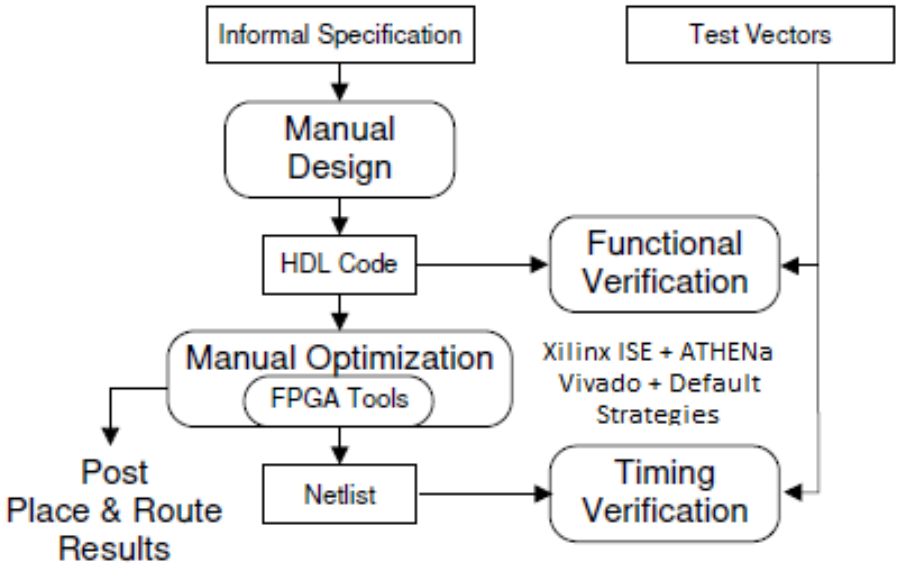


Figure 2.1: Traditional Hardware Development and Benchmarking Flow

than expected, e.g., in terms of the number of clock cycles required to process a single input block, the HLS-ready C code must be further optimized, and possibly extended with additional pragmas.

2.2.2 Test Case and Its RTL Implementations

As our test case, we have selected four Round 2 SHA-3 candidates: BMW, CubeHash, ECHO, and Luffa.

An algorithmic background on hash functions is introduced in [17]. All SHA-3 candidates and their evaluation is comprehensively covered at [18]. This website contains also useful information regarding the history of the SHA-3 competition and the standardization effort after announcing the winner. Finally, the submission packages of all candidates, including their detailed specification, reference and optimized software implementations in C, and a comprehensive set of test vectors, are available as well. These submission packages constitute a very important starting point for our study.

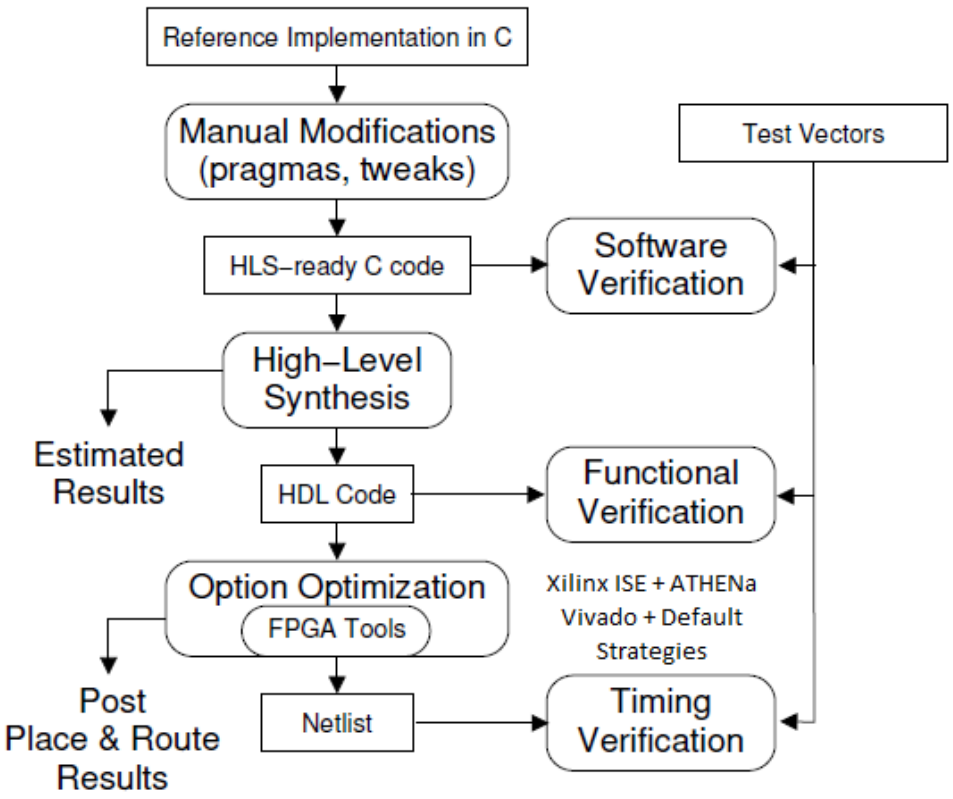


Figure 2.2: HLS-Based Development and Benchmarking Flow

In [19], [20] and [21] the results of high-speed RTL implementation of a number of SHA-3 candidates are reported. However, the main optimization target is throughput, rather than the throughput to area ratio, as in our study. In [22] a compact implementation of ECHO, targeting minimum resource utilization, is reported.

To provide a fair and reliable reference for comparison, the RTL designs used in our study are based on [23] and [4].

These designs were selected for the following reasons:

1. Source code of these designs in RTL VHDL is easily accessible to the public [24]. This availability allows everybody to easily replicate the obtained results.

2. Detailed diagrams of each design are also available in public domain [24]. The availability of these diagrams allows other designers to easily understand each specific hardware architecture, without analyzing the source code, and thus saving valuable time.

3. Uniform and realistic interface. The aforementioned studies utilized a standard FIFO-based interface, which can be easily adapted to support any bus-based interface for a system-on-chip, e.g., the AXI-4 Stream, interface.

4. Standalone. The reference designs are completely self-sufficient. They require the minimum number of external control signals.

2.2.3 Design Approach

The reference C code is based on the submission packages available from the SHA-3 contest website [27]. Each reference C implementation is modified to generate HLS-ready C code and then verified using C-Simulation in software. In the next step, the HLS-ready C code is given to the Vivado HLS and the tool converts it to VHDL code. This VHDL code is then simulated for functional correctness by Vivado HLS Co-simulation. If the results are incorrect, or the number of clock cycles and resource utilization, which is provided in the synthesis report, are too high, the HLS-ready code needs to be modified, and the entire process repeated. When the best results are achieved by the tool, the generated VHDL code is verified again using the universal testbench and the interface provided in [25]. This Interface is shown in Fig. 2.3. The simulator which is used for final verification is Xilinx ISim. The interface and the communication protocol used in this study are based on the designs proposed in [26]. As we are operating at full speed, the input and output units must operate at the same time when a cryptographic core is hashing. The current HLS tool is not yet capable of generating a design that can perform these tasks automatically. As a result, these modules need to be connected manually at the top-level.

The top-level diagram of all crypto cores is shown in Fig. 2.2. Each crypto core is

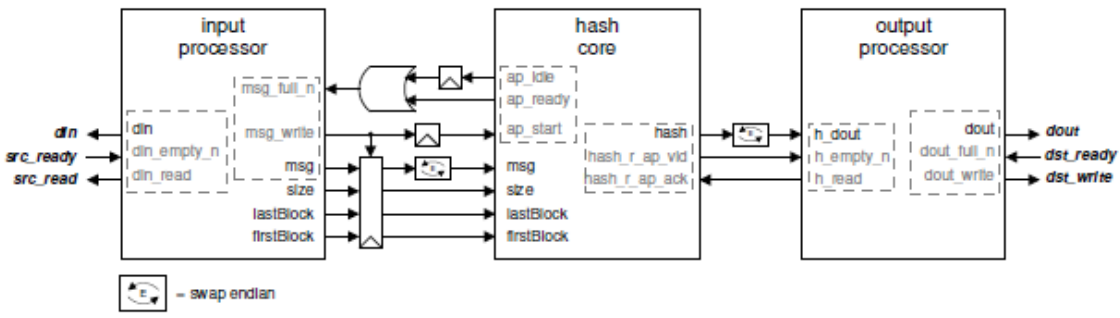


Figure 2.3: Top-level diagram

comprised of three primary modules, input processor, hash core and output processor. Each module is generated using HLS tools independently, using a separate HLS-ready C code, in order to allow them to operate concurrently. The rectangular regions within each module represent groups of signals that are generated by the HLS tool from the same input. For instance, din port of input processor generates din empty and din read signals. These generated signals are the product of the INTERFACE pragma. Communications between modules in the critical area are registered to improve timing. Endianness of incoming and outgoing data for the hash core are swapped for correct operation. This is because when ARRAY RESHAPE pragma is used to create a one dimensional array from a multi-dimensional array, a data block is formed based on the word size of the original array in the little endian format. For instance, forming a 128-bit data block from an integer array of size four (int A[4]) would have the following order: A[3] A[2] A[1] A[0]. As a result, when input data arrives in the big-endian format, endianness must be adjusted accordingly.

2.2.4 HLS Optimization Methodology

This section describes our HLS optimization methodology, based on [8]. The aim of this optimization is to infer an HLS-generated RTL code that has performance as close as possible to the manually developed RTL code. Baseline Implementation. The first step involves

creating an HLS-ready baseline implementation from the reference software. In this implementation, any variables and buffers that rely on dynamic allocation must be replaced by their static versions. Functions must be clearly declared and carefully controlled via the use of the INTERFACE pragma, in order to properly model the behavior of input and output ports of the targeted design. In this study, the model of our top-level function, translated to Hash Core (as defined in [27]), utilizes a previously established communication. The core receives a single block of data at a time, as opposed to a stream of blocks. This mode allows a more accurate report from the HLS synthesis tool (in terms of clock cycles) and a finer control in the optimization process detailed in the later sections. Code refactoring. Function reuse can be considered as one of the most critical factors in achieving an optimized design that has a performance comparable to RTL-based design. In software, repetition of a function call does not drastically affect performance or memory utilization of any program. However, unless an appropriate action is taken, a C synthesizer can treat each function call as a separate instantiation of a hardware module. As a result, multiple function calls can significantly increase the design area and critical path.

One of the approaches to mitigate this effect is to add a limit on the resource allocation for a specific function, in order to limit its reuse. The RESOURCE pragma facilitates this process. However, this approach requires further fine-tuning in order to reach a similar performance as the performance of the RTL-based design. While this fine-tuning is not a problem in a typical design environment, where the target clock frequency and area are known, it can be more difficult to apply for benchmarking purposes, where the expected clock period and resource utilization are hard to predict.

Rewriting source code in order to minimize the number of function calls is an alternative approach to mitigate an increase in area and, consequently due to routing congestion, a reduction in speed associated with the function call repetitions. While this approach requires more effort, it allows a finer control of the design, which can help to produce a more consistent results for benchmarking purposes. One of the most straightforward examples can be observed in case of the use of the round function, as shown in Fig. 2.4(a).

The original code includes six function calls to a round operation. While this approach is perfectly fine in software, it replicates the round operation in hardware, resulting in a longer critical path and higher resource utilization. A significant optimization can be accomplished by simply calling such a function in a loop, as shown in Fig. 2.4(b), so that only one instance of the corresponding module appears in hardware. In addition, as indicated in Fig. 2.5, if and else statement can be used instead of multiple function calls for different inputs to avoid multiple component instantiation in hardware.

```
oneround(A,S,0);
oneround(B,A,1);
oneround(A,B,2);
oneround(B,A,3);
oneround(A,B,4);
oneround(S,A,5);
```

(a) Regular C code before modification

```
for (round=0; round<6; ++round)
{
    oneround(S, S, round);
}
```

(b) HLS-compatible modification

Figure 2.4: Example of code modification for resource reduction purposes

```
//Beore modification
  if (lastBlock == 1)
  {
      rrounds(chain_tmp, 2);
  }
  else
  {
      rrounds(chain_tmp, 1);
  }
```

(a)

```
//After modification
  if (lastBlock == 1)
  {
      x_round = 2;
  }
  else
  {
      x_round = 1
  }
  rrounds(chain_tmp, x_round);
```

(b)

Figure 2.5: Function reuse utilizing if and else statement

Data storage. The next step is optimizing data types and sizes of arrays, and providing the tool with directives on how these arrays should be translated into hardware. Vivado HLS synthesis tool supports a flexible base data type, uint[size], where [size] is replaced by

the desired data size. Utilizing a correct size for our needs can significantly increase design efficiency. For instance, if an integer array only uses the first four bits, it might be better to define data type of that array as uint4 instead of integer type.

The speed at which data can be accessed, or bandwidth, is determined by its type and/or array dimension. The operation on an array of data can be significantly hampered if the design is not able to access different elements at the same time. This problem can be mitigated by using the ARRAY RESHAPE directive to re-partition an array to different dimensions without changing the underlying code. An array is synthesized into a memory if it is not partitioned into a complete one-dimensional data. An array reshaping example is shown in Fig. 2.6.

```
void luffa( uint32 data_in[8], uint32 hash[hash_len], uint1 firstBlock, uint1 lastBlock)
{

    uint32 static chain [chain_size];
    uint32 chain_tmp[chain_size];
    uint32 b_tmp[hash_len];

#pragma HLS ARRAY_RESHAPE variable=data_in   complete dim=1
#pragma HLS ARRAY_RESHAPE variable=hash      complete dim=1
#pragma HLS ARRAY_RESHAPE variable=b_tmp     complete dim=1
#pragma HLS ARRAY_RESHAPE variable=CNS       complete dim=1
#pragma HLS ARRAY_RESHAPE variable=IV        complete dim=1
#pragma HLS ARRAY_RESHAPE variable=chain     complete dim=1
#pragma HLS ARRAY_RESHAPE variable=chain_tmp complete dim=1
```

Figure 2.6: Example of ARRAY RESHAPE directive

Loop Optimization. In order to fully realize the potential of the design after data storage has been optimized, all loops of the program must be optimized as well. The UNROLL pragma is the best directive to realize this potential. As indicated in Fig. 2.7, this directive informs the tool to unroll the operations within a loop so they can be executed in parallel, thereby increasing the overall throughput of the loop.

A memory can either be a ROM or RAM depending on the pragma specified. A ROM inference can be done via the RESOURCE pragma, using an asynchronous ROM (ROM 1P

```
for (k = 0;k < 8;++k)
{
#pragma HLS UNROLL
    chain_tmp[k] ^= BYTES_SWAP32 (data_in[k]);
}
```

Figure 2.7: Example of Loop Optimization directive

1S) core, which is one of many hardware types that can be specified. This type is particularly useful when the designer does not want to the tool to implement a given storage component using BlockRAM. The change to asynchronous ROM can significantly reduce the design latency. The limitation for a LUT-based RAM is also an issue. While the tool is capable of selecting a correct RAM resource, it has a problem with scheduling the right operation. In particular, an additional clock cycle is required if there is a read from this unit. The tool requires all RAM read operations to have one clock cycle delay. This feature can substantially increase design latency when a main loop, which is responsible for majority of latency, needs to access a memory element. While RESOURCE pragma, using ROM (ROM 1P 1S), decreases latency, the tool has limitation in multiple instantiation of one block in parallel automatically. For example, as shown in Fig. 2.8(a), the tool instantiates 1 instance of sbox in hardware and this code takes 5 clock cycles to execute. Therefore, the only way to do this task in parallel and in one clock cycle is to use Fig. 2.8(b) tweak.

2.3 Features of the Implemented Hash Algorithms and their Hardware Architectures

Four Round 2 SHA-3 candidates investigated in this project are BMW, CubeHash, ECHO, and Luffa. The major features and parameter values used consistently in our RTL and HLS implementations of these hash functions (Number of rounds, Block size, Hash size and State size) are represented in Table 2.1. In particular, the Hash size has been set to

256 bits for all algorithms. Our primary optimization target is the best Throughput to Area ratio, where Throughput is defined as Throughput for long messages, and Area is expressed in terms of the number of LUTs (Look-Up Tables) of Xilinx FPGAs. This choice has multiple advantages. First, it is practical, as hardware cores are typically applied in situations, where the size of the processed data is significant and the speed of processing is essential. Otherwise, the communication overhead associated with using a hardware accelerator dominates the total processing time, and the cost of using dedicated hardware (FPGA) is not justified. Optimizing for the best ratio provides a good balance between the speed and the cost of the solution, and typically leads to the most natural basic iterative architecture. In particular, this choice leads to optimum values for the number of clock cycles per message block, and thus determines the formulas for throughput.

The numbers of clock cycles per one block of message (#Cycles/Block) for the RTL and HLS implementations are summarized in Table 2.2. For the majority of implemented Hash algorithms, the automatically generated Controller (initially described in C) tends to be sub-optimal, compared to the manual design using RTL VHDL. This is because the HLS-generated unit is not capable of supporting an overlap between the completion of the last round and reading the next input block. Moreover, one additional clock cycle is often required to initiate processing of the subsequent data block. As a result, the throughput of the HLS-based design tends to be lower than the throughput of the manual design, even if they both operate at the same clock frequency. However, there are some exceptions, for example, in case of ECHO #Cycles/Block in RTL-based approach and HLS-based approach are exactly the same due to slightly different architecture. ECHO has a very high resource utilization and long critical path compared to other algorithms due to its utilization of 16 AES rounds which are working in parallel. As a result, the RTL-approach designer tried to reduce the critical path by registering the output in specific location with the cost of growth in #Cycle/Block, but in our experiment this approach did not improve the HLS result and is avoided during the design process. In addition, the synthesis process in the Vivado HLS environment for the ECHO algorithm takes about one hour after each modification that

inlines functions in a single scope, which avoids communication delay between separate functions and achieves results similar to RTL implementation. The synthesis process for high level synthesis takes about 3 to 10 minutes in case of less complicated algorithms.

Table 2.1: Parameters of Hash functions

Algorithm	**Num. of Rounds**	**Block Size**	**Hash size**	**State Size**
Luffa	8	256	256	768
CubeHash	16	256	256	1024
BMW	1	512	256	512
ECHO	8	1536	256	512

Table 2.2: HLS and RTL implementations parameters

Algorithm	**Num. of Rounds**	**Cycles/ Block RTL**	**Cycles/ Block HLS**
Luffa	8	9	11
CubeHash	16	16	18
BMW	1	2	1
ECHO	8	26	26

2.4 Results and Discussion

The ranking of all implemented algorithms based on resource utilization (Area) for both RTL and HLS implementation is shown on Fig. 2.9. As we can observe, the ranking of the algorithms using HLS implementation is exactly the same as RTL implementation. CubeHash has the smallest area and Luffa, BMW and, ECHO are on the second, third and fourth place respectively. All candidates are ranked based on Throughput in Fig. 2.10. The results indicate that HLS and RTL results again have exactly the same ranking in case

of throughput. ECHO has the highest throughput and Luffa, BMW and CubeHash are in subsequent positions, respectively.

The throughput over area results of all implemented algorithm for RTL and HLS-based design flows are summarized in Fig. 2.11 and Fig. 2.12. Each algorithm is represented using a different shape marker, with lines across the markers helping to compare the throughput to area ratios. The higher the gradient of the line, the more efficient the corresponding algorithm is. These diagrams demonstrate a very good correlation between the HLS and RTL results and the ranking is the same in both cases. The details of all results and the corresponding ratios are listed in Tables 2.3, 2.4 and 2.5. The throughput formulas for the RTL and HLS-based approach is given by equation 1. All results are generated for Xilinx Virtex 5.

Table 2.3: Results for the HLS approach

HLS				
	Freq.	**TP**	**A**	**TP/A**
Luffa	248.76	5,789	1,283	4.51
CubeHash	247.34	3,517	1,034	3.40
BMW	10.76	5,511	5,042	1.09
ECHO	176.80	10,445	5,767	1.81

Table 2.4: Results for the RTL approach

RTL				
	Freq.	**TP**	**A**	**TP/A**
Luffa	334.4	9,513	1,023	9.29
CubeHash	248.3	3,972	663	5.99
BMW	34.0	8,704	4,350	2.00
ECHO	203.8	12,094	4,888	2.47

Notations: Freq.: clock frequency, A: area in CLB slices, TP: Throughput in Mbits/s, TP/A: throughput over area ratio, RTL/HLS: ratio of the RTL result to the corresponding

Table 2.5: RTL/HLS Result Ratios

RTL/HLS				
	Freq.	**TP**	**Area**	**TP/A**
Luffa	1.34	1.64	1.25	2.06
CubeHash	1.00	1.13	0.64	1.76
BMW	3.16	1.58	0.86	1.83
ECHO	1.15	1.16	0.85	1.36

HLS result.

2.5 Conclusions

High-level synthesis offers a potential to allow hardware benchmarking in early stages of cryptographic contests. It can also be an effective method for gauging the hardware performance of early variants of a cryptographic algorithm during the design process by groups of cryptographers with limited experience in hardware design. Our case study based on the five final SHA-3 candidates has demonstrated the correct ranking for Xilinx Virtex5 FPGAs in terms of all major performance measures: frequency, throughput, area, and the throughput to area ratio. In order for the HLS-based design approach to be an effective replacement for the manual design approach, there are a few problems that need to be still overcome. These obstacles include: 1- generation of suboptimal control units by Vivado HLS tools. 2- Efficient and reliable generation of HLS-ready C code. 3- Wide range of RTL to HLS performance metric ratios.

```
const uint4 S0[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};

void subcrumb_1(uint32 a[8])
{
  #pragma HLS RESOURCE variable=S0 core=ROM_1P_1S

  convtobit32(a[3], a[2], a[1], a[0], sbox_in_0);

  for (i=0; i<5; i++)
  {
  #pragma HLS UNROLL
    sbox_out_0[i] = S0[sbox_in_0[i]];
  }
}
```

(a)

```
const uint4 S0_0[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};
const uint4 S0_1[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};
const uint4 S0_2[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};
const uint4 S0_3[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};
const uint4 S0_4[16] = {13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4};

void subcrumb_1(uint32 a[8])
{
  #pragma HLS RESOURCE variable=S0_0 core=ROM_1P_1S
  #pragma HLS RESOURCE variable=S0_1 core=ROM_1P_1S
  #pragma HLS RESOURCE variable=S0_2 core=ROM_1P_1S
  #pragma HLS RESOURCE variable=S0_3 core=ROM_1P_1S
  #pragma HLS RESOURCE variable=S0_4 core=ROM_1P_1S

  convtobit32(a[3], a[2], a[1], a[0], sbox_in_0);
  convtobit32(a[4], a[7], a[6], a[5], sbox_in_1);

  sbox_out_0[0] = S0_0[sbox_in_0[0]];
  sbox_out_0[1] = S0_1[sbox_in_0[1]];
  sbox_out_0[2] = S0_2[sbox_in_0[2]];
  sbox_out_0[3] = S0_3[sbox_in_0[3]];
  sbox_out_0[4] = S0_4[sbox_in_0[4]];

}
```

(b)

Figure 2.8: Multiple instantiation of ROM blocks in parallel automatically

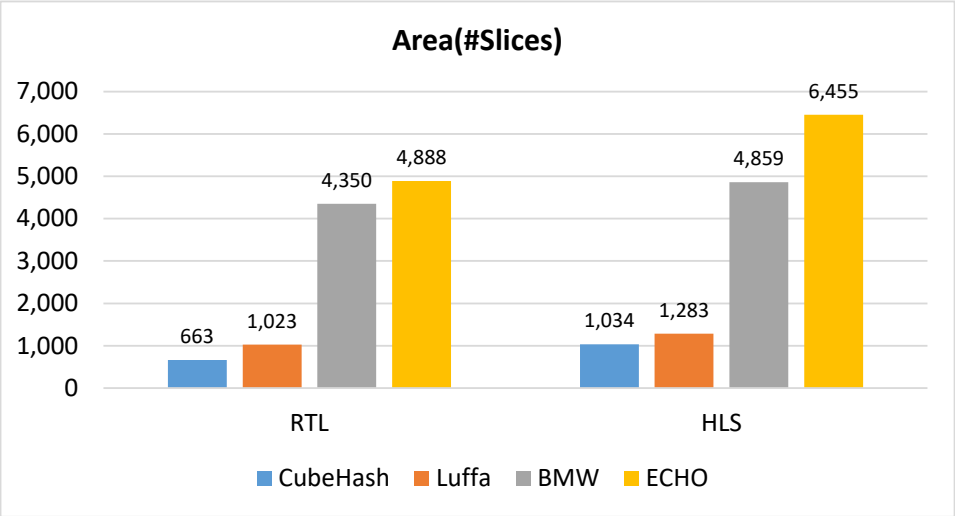


Figure 2.9: Resource utilization (Number of CLB Slices)

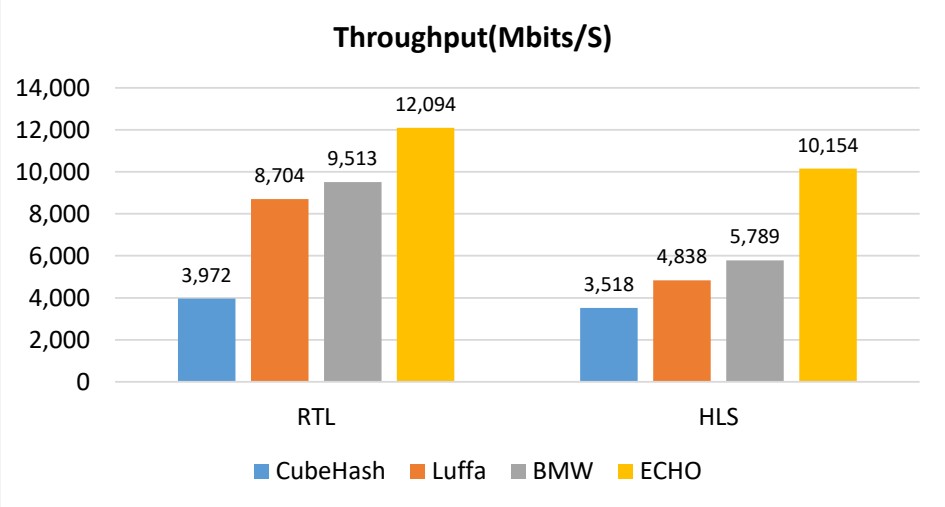


Figure 2.10: Throughput (Mbits/s)

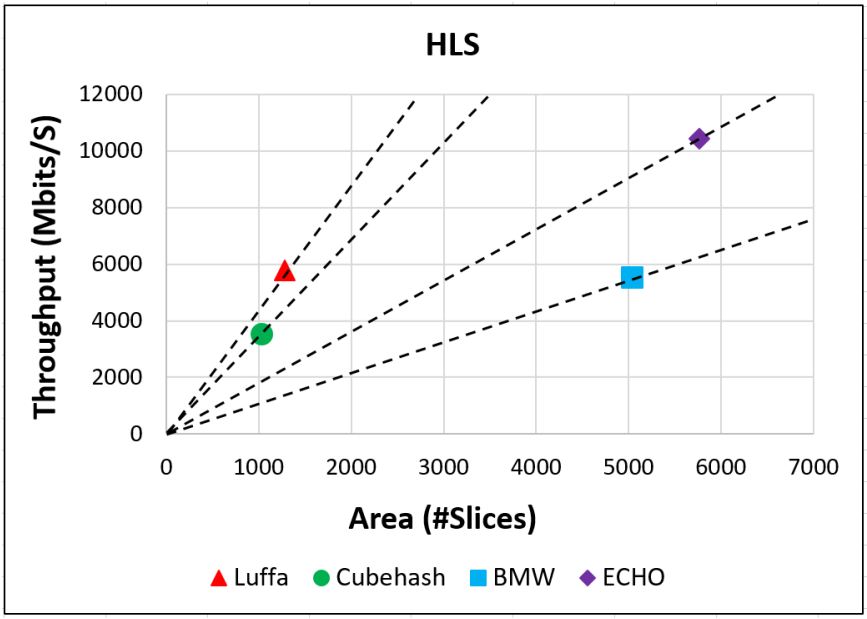


Figure 2.11: HLS Implementation: Throughput over Area

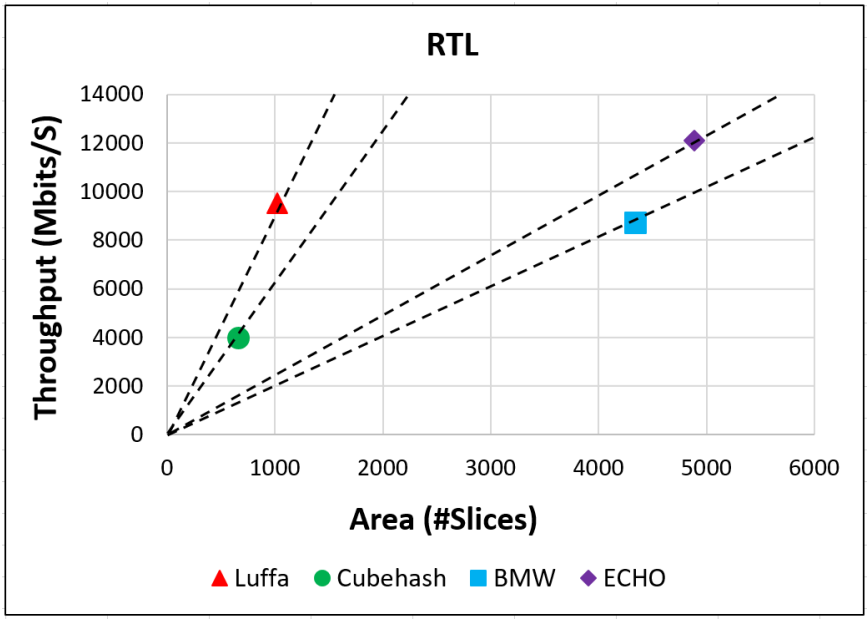


Figure 2.12: RTL Implementation: Throughput over Area

Chapter 3: Minerva

3.1 Previous Work

In [16] an open-source environment for fair, comprehensive, automated, and collaborative hardware benchmarking of algorithms belonging to the same class is presented. The main part of this environment is the ATHENa tool for optimization of tool options, requested clock frequency, and the starting point of placement. ATHENa provides similar capabilities for designers targeting FPGA devices from two major vendors, Xilinx and Altera. However, it works only with the previous-generation Xilinx CAD tool (ISE), which will not support Xilinx FPGAs beyond the Series 7 families (Virtex 7, Artix 7, Kintex 7).

Moreover, FPGA vendors by themselves have their own tools for the exploration of implementation options. One example is ExploreAhead [28] from Xilinx, which is a part of the high-level optimization tool called PlanAhead. PlanAhead is provided as a built-in option in Vivado Design Suite, the latest version of Xilinx CAD tools. ExploreAhead allows executing multiple implementation runs based on predefined or user defined strategies (understood as preselected values for a set of options). Additionally, it supports parallel runs on multi-core CPUs. Unlike ATHENa, which supports two vendors, PlanAhead works only with Xilinx FPGAs. Additionally, ATHENa is aimed at achieving the best possible performance (e.g., the best throughput/area ratio), while ExploreAhead and Vivado aim only at achieving the requested clock frequency.

[29] presents a tool called SUPERCOP, which expedites comparison of software implementations of cryptographic algorithms. This open source tool supports the choice of the best compilation options from thousands of different combinations. It also facilitates execution time measurements on multiple computer systems.

3.2 Environment

In this section we describe features of our new environment, called Minerva, that provides functionality similar to ATHENa for Xilinx Series 7 FPGAs and beyond.

In order to observe the behavior of the Vivado Design Suite in static timing analysis, synthesis and implementation were performed for 5 CAESAR Round 2 candidates. At first, the same requested clock frequency constraint was used for each algorithm. The target clock frequency was set to 333 MHz, and the theoretical achievable frequency was calculated based on WNS (Worst case Negative Slack), utilizing the following formula:

$$Minimumclockperiod = Targetclockperiod - WNS \tag{3.1}$$

In the next step, WNS results were generated for the requested clock frequency varying in range of -64 to +64 MHz of the reference frequency, with the precision of 1 MHz. It meant that we generated WNS results for 128 different target clock frequencies in order to observe a trend. Fig. 3.1, Fig. 3.2, Fig. 3.3, Fig. 3.4 and Fig. 3.5 show this trend for Joltik, AES–GCM, ICEPOLE, SCREAM and TriviA-ck, respectively. GraphGen function provided by Minerva accommodated the aforementioned process.

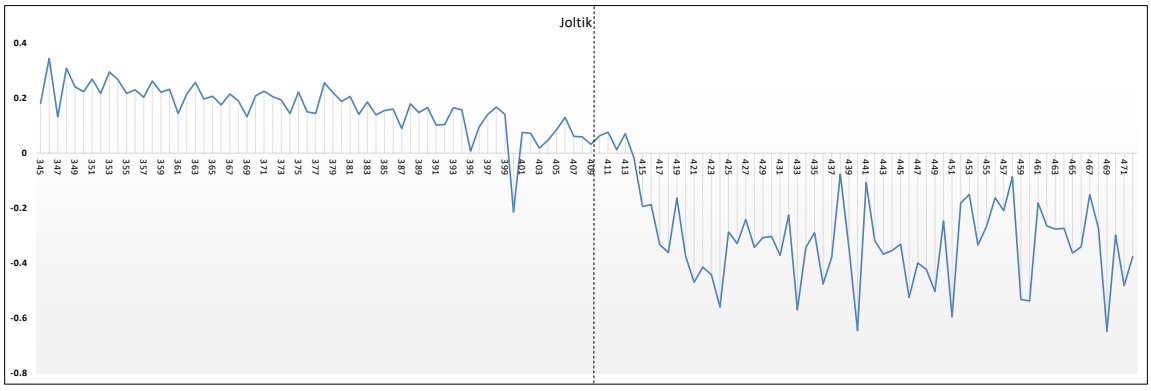


Figure 3.1: Graph generated for Joltik using results provided by GraphGen

As we can observe in Fig. 3.1, Fig. 3.3, Fig. 3.4 and Fig. 3.5, we have fluctuations

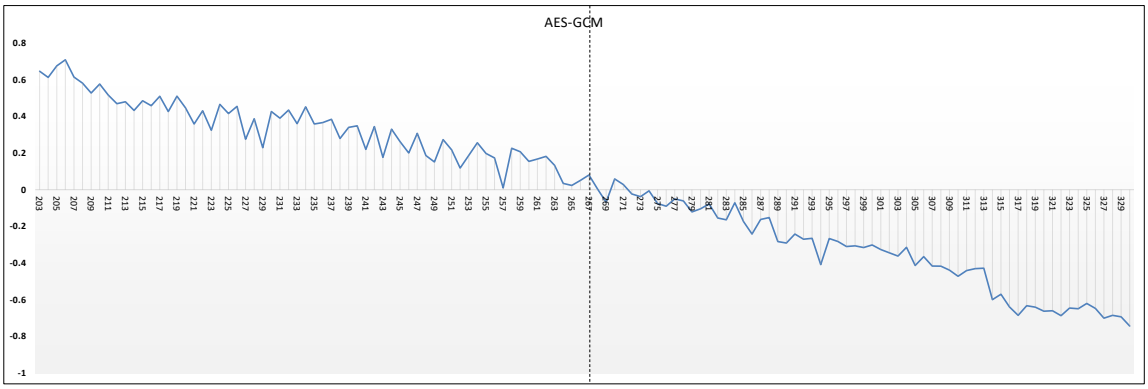


Figure 3.2: Graph generated for AESGCM using results provided by GraphGen

around the calculated reference clock frequency. This fluctuation is much higher in case of ICEPOLE and SCREAM. As a result, it would be very hard to find the actual maximum clock frequency without automation. We have lower fluctuations for AES-GCM, Joltik and TriviAck. Based on Xilinx documentation [30], the only acceptable target frequency is the one that gives us positive slack. Therefore, based on the aforementioned graphs we cannot rely on the above equation to calculate the actual maximum clock frequency. Instead, we need a more complex procedure. In addition, these results are generated using only default options of Vivado for all implementation steps, such as mapping, placing and routing. The Vivado Design Suite ships with 25 predefined optimization strategies, which can be used to achieve higher maximum frequency and more optimized design. Hence, incorporating all of these strategies leads to an even more tedious process.

One way is to use binary search algorithm to find the maximum frequency in a given frequency range. However, there are two problems associated with this method: 1- We cannot easily cover 25 optimization strategies. 2- based on the fluctuations observed in the generated graphs, different results will be achieved for different input ranges. Also, it is possible that none of them will be the actual maximum clock frequency. As a result, we equip Minerva with the heuristic algorithm aimed at addressing this problem.

Minerva is used to execute Vivado in batch mode, utilizing the Vivado batch mode Tcl

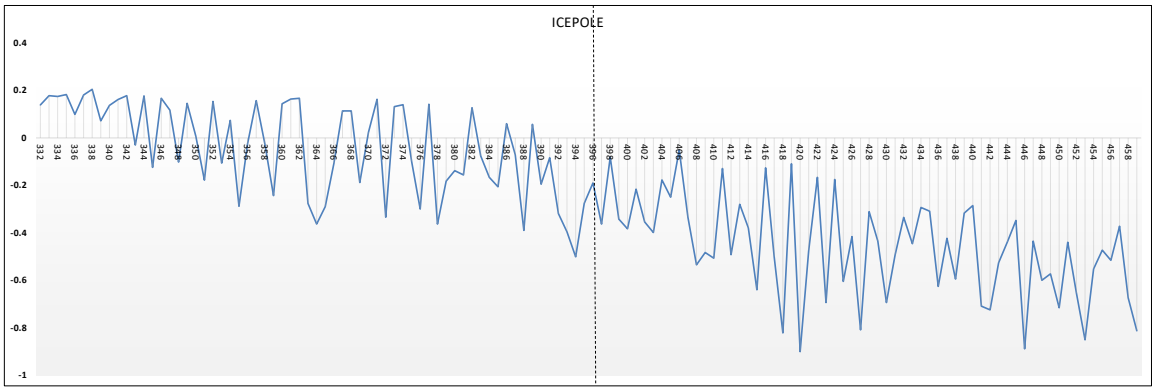


Figure 3.3: Graph generated for ICEPOLE using results provided by GraphGen

scripts provided by Xilinx. An XML-based Python program is used to manage runs. This program launches Vivado with Tcl scripts that are dynamically created during run-time and later modified to perform each step of the optimization algorithm. Minerva is designed to be used to automate the task of finding optimized results for each folder of a source code repository, and it works with any device that Vivado supports.

3.3 Design Flow

Minerva supports multiple frequency search algorithms and it has the capability to add new algorithms in the future. Minerva frequency search (MinervaFreqSearch) is designed specifically to find the maximum frequency achievable by a given hardware design. MinervaFreqSearch function receives the following parameters as input:

1. f0 (frequency 0) and fn (frequency n): these are the lower bound and upper bound frequencies of the range that we span to find the maximum frequency. These values can be updated during the run time.

2. n: indicates the number of runs to be performed in parallel. Minerva can run on multiple CPU cores and take advantage of multithreading.

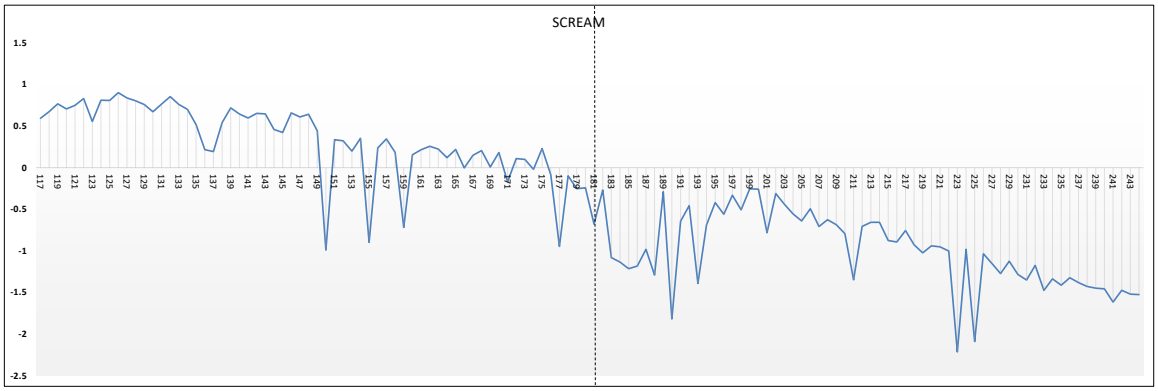


Figure 3.4: Graph generated for SCREAM using results provided by GraphGen

3. p: represents the number of optimization strategies to be considered during the search.

4. r (precision range size): is the maximum number of frequency targets (higher than the last achieved maximum clock frequency) to be explored. If we achieve positive slack for a frequency in this range, we will continue the search, otherwise we will terminate the process. The recommended value for r is 12. Using lower values is possible, but it may cause achieving lower frequency than actual maximum frequency. On the other hand, using higher values leads to increase in the execution time.

This function generates an output report that contains the following information:

1. WNS result for all test cases with the corresponding optimization strategy ID and target clock frequency.

2. WNS and Area results for all target frequencies with positive slack.

3. Maximum frequency and the corresponding Area in number of LUTs and slices, and optimization strategy ID.

4. Maximum frequency/LUTs and the actual values for frequency, number of LUTs, number of Slices and the optimization strategy ID for this test case.

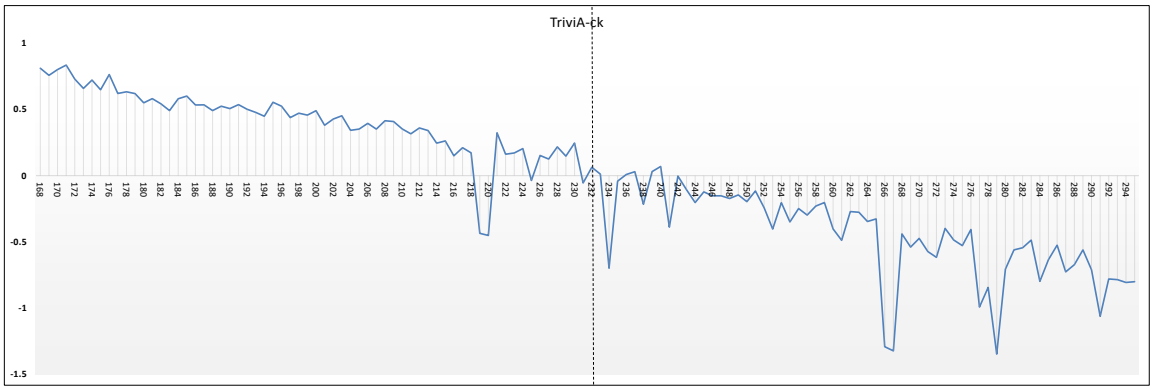


Figure 3.5: Graph generated for Triviack using results provided by GraphGen

5. Execution time.

Fig. 3.7(a), Fig. 3.7(b), Fig. 3.7(c), Fig. 3.7(d) and Fig. 3.7(e), completely describe how MinervaFreqSearch algorithm works. Each column illustrates one requested clock frequency value, and square blocks in that column correspond to optimization strategies. Each square block represents one test case with the optimization strategy ID mentioned inside it. Colors of these blocks are green or red, indicating the positive or negative WNS, respectively. The runs that execute in parallel at each step are represented using dotted boxes.

Fig. 3.7(a) shows the first step in MinervaFreqSearch algorithm and the legend represents input parameters related to this example (f0=50, fn=200, n=8, r=8 and p=8). In the first step, the given frequency range (50 to 200) is divided by r+1 to have 8 frequencies including 50 and 200, with the same distance between each other, as shown in Fig. 3.7(a) Freq axis. Then, WNS result is generated for all of these 8 target frequencies and the default optimization strategy. It is feasible to run all of these target frequencies at the same time, as n is equal to 8 in this example. After WNS results are generated, if the upper bound frequency (fn) gives us positive slack, we will update f0 and fn values using the formula 3.2 and 3.3, and will repeat the previous process (step forward).

$$f0(new) = fn(old) + 1 \tag{3.2}$$

$$fn(new) = fn(old) + 100 \tag{3.3}$$

If all of the first 8 target clock frequencies give us negative slack, we will step backward for 100 MHz frequency range. So f0 and fn will be updated using formulas 3.4 and 3.5, and the first step will be repeated.

$$f0(new) = fn(old) - 100 \tag{3.4}$$

$$fn(new) = f0(old) - 1 \tag{3.5}$$

The aforementioned process leads to finding the maximum frequency less than fn that gives us positive slack using only default optimization strategy. As we can observe in Fig. 3.7(a), in the first step, positive slack was achieved for fn (200 MHz). Hence, we step forward and update f0 and fn to 201 and 300 MHz respectively, Fig. 3.7(b). As shown in this figure, 243.9 MHz is the highest frequency that leads to positive slack with default optimization strategy. At this point, the optimization runs are started for the remaining frequencies higher than 243.9 MHz in this range. In this example 258.2 MHz, with optimization strategy number 3 has positive slack, so the maximum frequency will be updated to 258.2MHz. In case of higher frequencies, all 8 optimization strategies failed. Therefore, 258.2MHz would be our starting point to begin the next step of frequency search considering 8 optimization strategies and the precision of 1MHz. Next step is illustrated in Fig. 3.7(c). In this step we go forward by 1MHz. As soon as we find a frequency with positive slack, the lower frequencies and the remaining optimization strategies corresponding to these frequencies are eliminated. The aforementioned procedure will be continued until 8 (precision range size) consecutive frequencies fail to provide positive slack for all possible optimization strategies (8 in this example), as shown in Fig. 3.7(d) and Fig. 3.7(e). Therefore, the maximum frequency in this example is 269MHz, with the optimization strategy number 4.

3.4 Result

Vivado Design Suite 2015.1 has been used for result generation. The target device has been set to the largest and fastest Virtex 7, xc7vx485- tffg1761-3. Binary search is done by considering only the default optimization strategy and Minerva frequency search is configured using the following values: n = 16, p = 23, r = 12 and the input range is [100, 500] for all candidates.

Table 3.1 presents detailed values of the maximum clock frequency and area generated using Minerva for 21 Round 2 CAESAR candidates. The second and the third columns show frequency in MHz and area in the number of LUTs, respectively, obtained by utilizing Minerva frequency search and targeting Throughput as optimization criteria. Fourth and fifth columns similarly report the results, but in case of Throughput over Area ratio as the main optimization target instead. For most of the candidates (17), the obtained result is similar for both cases. Maximum frequency is reduced in case of AEZ, Joltik, Minalpher and TriviA-ck to achieve better Throughput to Area ratio. However, this reduction is less 13MHz for all aforementioned candidates. Last two columns contain maximum frequency and number of LUTs obtained by applying the binary search algorithm with a single optimization strategy.

Fig. 3.6 illustrates the ratio of results obtained using Minerva frequency search vs. Binary search in terms of Throughput, Area and Throughput/Area. As we can see, Throughput/Area ratio has improved by almost 40% for ICEPOLE and more than 20% in case of AEZ. The order of candidates is based on the decreasing improvement in terms of Throughput/Area ratio. This metric is improved by more than 10% for the next 6 candidates. As expected, algorithms which have more fluctuations around the reference frequency in the previously generated graphs, such as ICEPOLE (Fig. 3.3), SCREAM (Fig. 3.4) and Triviack (Fig. 3.5), take advantage of Minerva frequency search much more than the stable ones (8-38% vs. less than 5%).

Table 3.2 presents the execution time for the Minerva frequency search and the binary

Table 3.1: Detailed values of the maximum clock frequency (MHz) and area (number of LUTs) generated using Minerva for 21 Round 2 CAESAR candidates

Algorithm	**Minerva Opt. Freq.**		**Minerva Opt. Freq/LUT**		**Binary search**	
	Freq. [MHz]	#LUTs	Freq. [MHz]	#LUTs	Freq. [MHz]	#LUTs
AESCOPA	278	7,707	278	7,707	270	7,702
AEZ	384	5,160	374	5,018	315	5,101
ASCON	444	1,557	444	1,557	427	1,542
CLOC	257	3,848	257	3,848	247	3,816
Deoxys	369	3,317	369	3,317	356	3,313
HS1-SIV	236	8,105	236	8,105	226	8,066
ICEPOLE	442	5,130	442	5,130	357	5,677
Joltik	439	1,606	427	1,558	414	1,592
Minalpher	243	7,953	237	7,635	203	7,326
NORX	206	4,474	206	4,474	198	4,371
OCB	350	4,486	350	4,486	335	4,487
OMD	297	4,628	297	4,628	259	4,586
PAEQ	322	8,373	322	8,373	280	8,334
PI-Cipher	214	3,906	214	3,906	192	3,838
POET	247	7,490	247	7,490	216	7,380
PrimatesGIBBON	214	1,892	214	1,892	185	1,827
PrimatesHANUMAN	209	1,840	209	1,840	195	1,796
SCREAM	186	2,630	186	2,630	177	2,764
STRIBOB	380	4,687	380	4,687	370	4,648
TRIVIA-ck	255	2,593	254	2,578	233	2,571
AES-GCM	277	3,114	277	3,114	272	3,097

search, respectively. As shown in this table, similarly to the Throughput/Area ratio improvement, Minerva frequency search run time depends on the corresponding candidate's graph stability. AES-GCM with the most stable graph has the lowest run time (3 hour and 20 minutes) and ICEPOLE with the most fluctuated graph has the highest execution time (12 hours and 20 minutes). In addition, Minerva run time has direct relation with n (number of runs in parallel) which is 16 in this case. One the other hand, the time of the binary search is very consistent for all 11 algorithms.

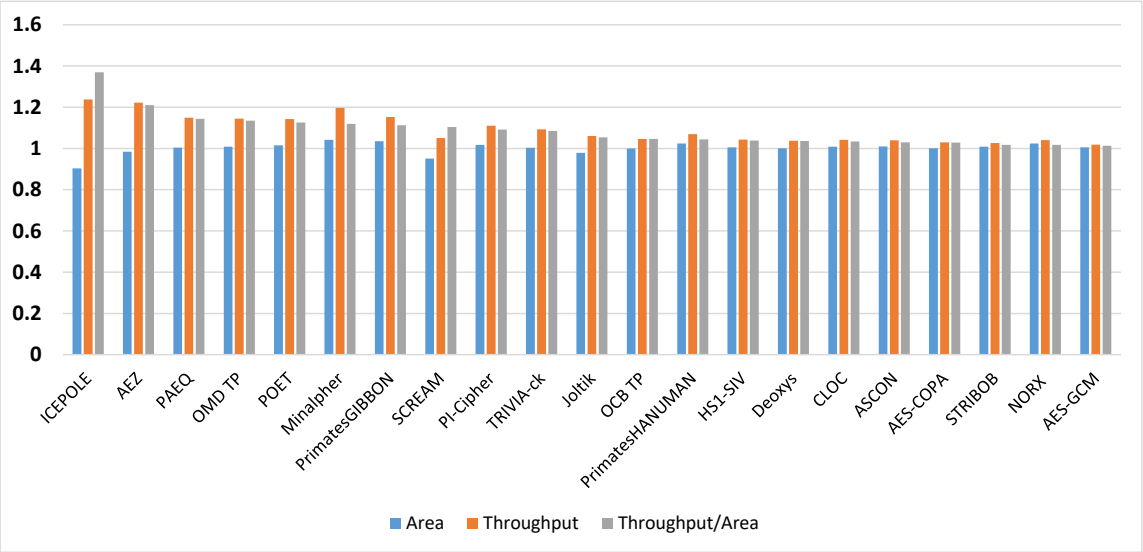
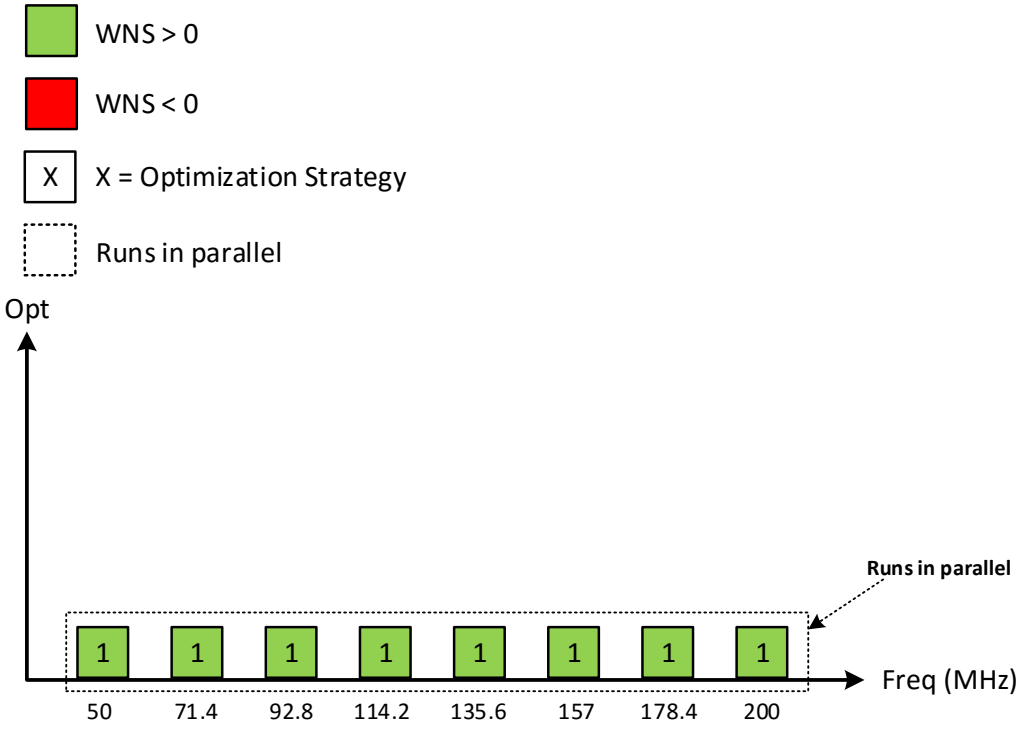


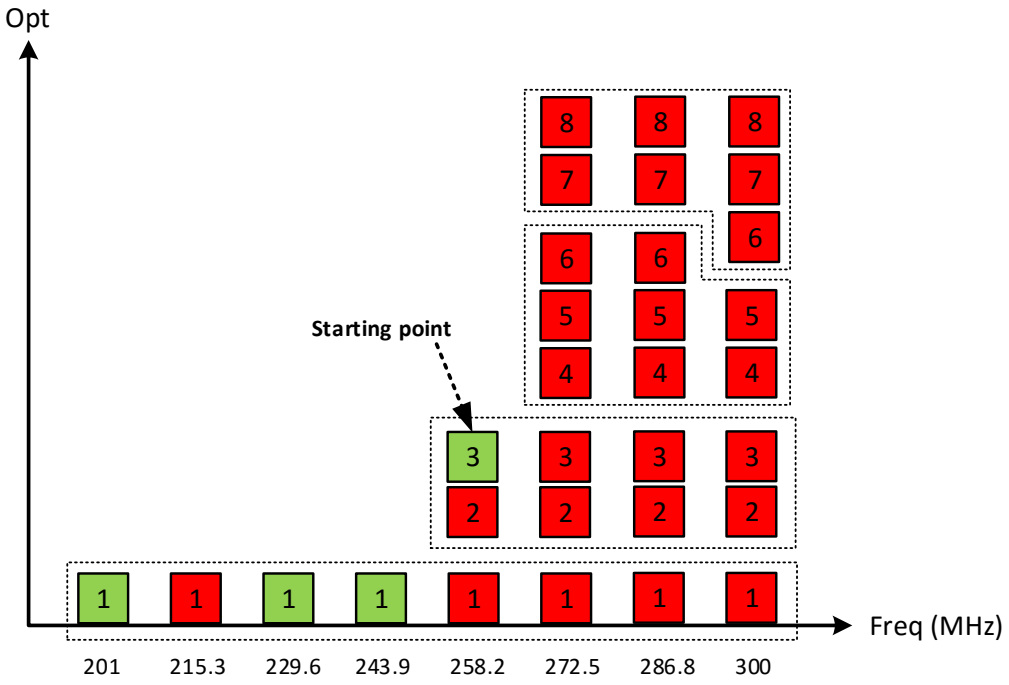
Figure 3.6: Ratio of results obtained using Minerva frequency serach vs. binary search

Table 3.2: Minerva frequency search and binary search run time for 11 CAESAR candidate

Algorithm	**Minerva Run time hrs:min**	**Binary Search Run time hrs:min**
AES-COPA	6:10	1:00
CLOC	6:5	1:35
ICEPOLE	12:20	1:00
Joltik	5:20	0:45
NORX	6:00	1:15
OCB	5:15	1:20
OMD	5:45	0:50
SCREAM	6:30	1:10
STRIBOB	6:00	1:30
TRIVIA-ck	5:20	1:04
AES-GCM	3:20	1:04

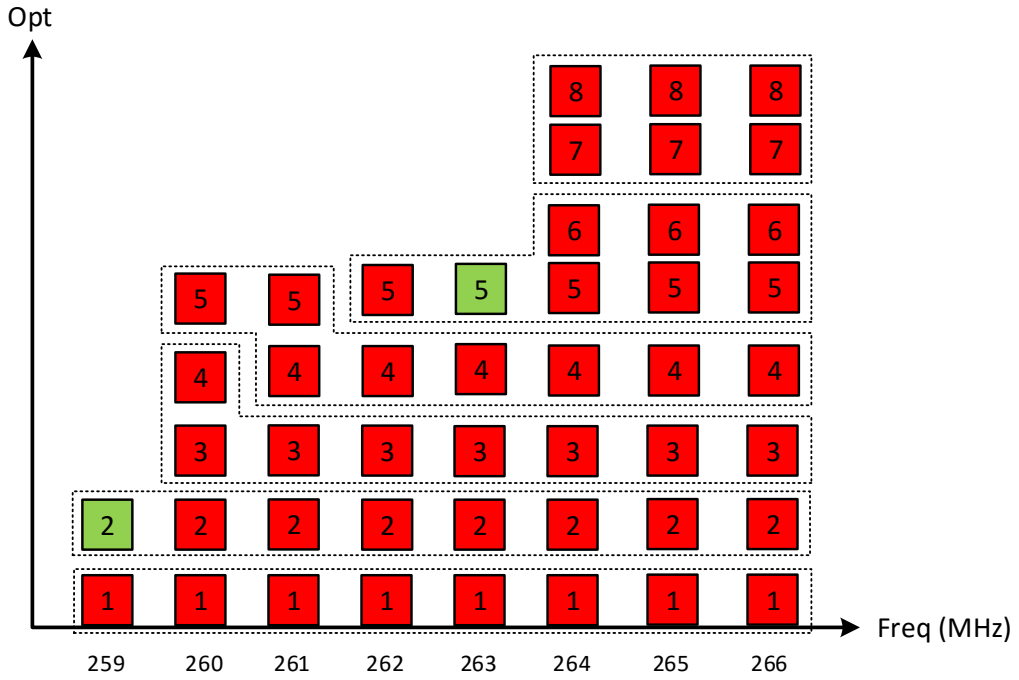


(a) Step1

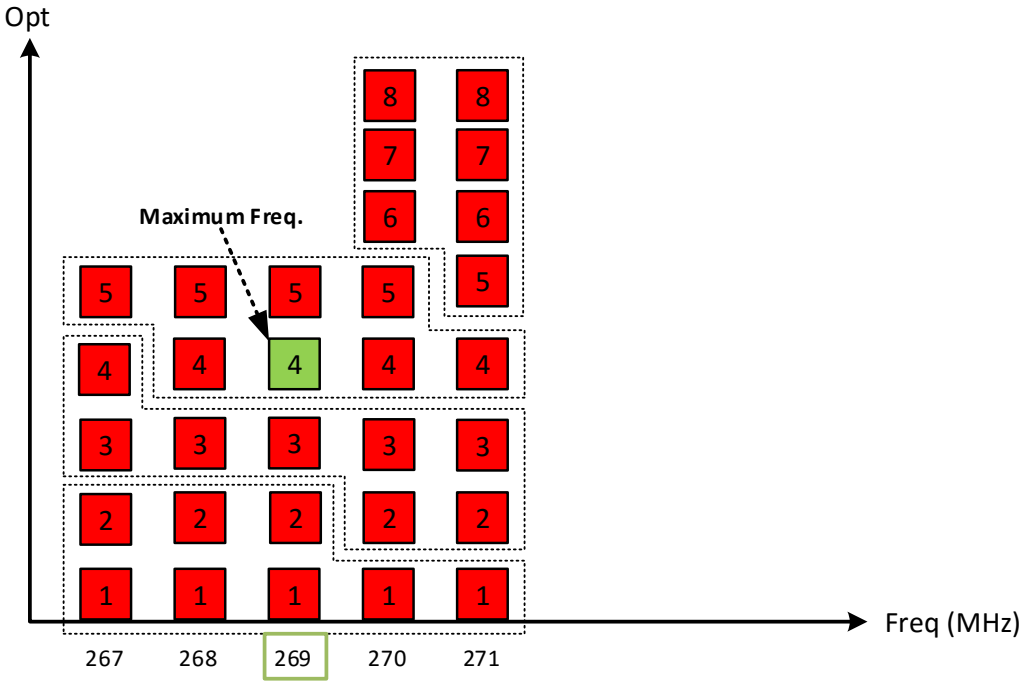


(b) Step2

Figure 3.7: Graphical representation of Minerva frequency search algorithm

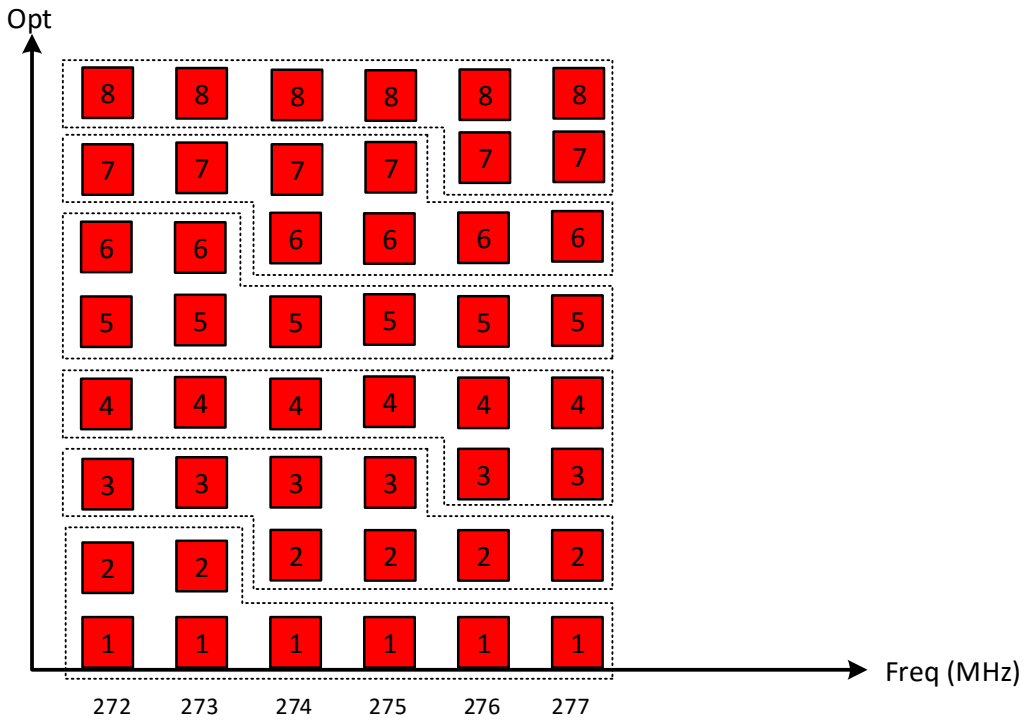


(c) Step3



(d) Step4

Figure 3.7: Graphical representation of Minerva frequency search algorithm



(e) Step5

Figure 3.7: Graphical representation of Minerva frequency search algorithm

Chapter 4: A Zynq-based Testbed for the Experimental Benchmarking of Algorithms Competing in Cryptographic Contests

4.1 Previous Work

Experimental benchmarking of cryptographic algorithms has been performed previously on different platforms other than Zynq. In [31], maximum frequency of SHA-256 has been measured experimentally using the SLAAC-1V board based on Xilinx Virtex VCV 1000. In [32] and [33], an experimental measurement of the hardware performance of 14 round 2 SHA-3 candidates is performed using the SASEBO-GII FPGA board. The investigated implementations are run at their maximum clock frequency reported by the CAD tool. Hence, no investigation of higher clock frequencies is performed.

In [34], full hardware implementations of all Round 2 candidates in hardware for all specified message digest variants and their post-place-and-route implementation results for a Virtex-5 FPGA is provided. The author compared the efficiency of hash functions in terms of throughput per unit area. In addition, a universal hardware wrapper interface was used in this work that produces the padding scheme required for a particular hash function as well as providing the interface to the outside world. This wrapper completely described in [35]. Area and Maximum frequency results represented for design with and without wrapper, but no experimental measurements were performed in this work.

In [36], the experimental evaluation of SHA-2 and five Round 3 SHA-3 Candidates was performed using a standard-cell ASIC realized using 65nm CMOS technology. The authors combined two sets of implementations, developed using different performance targets, and implemented them on one ASIC, with a common input/output (I/O) interface.

Area, throughput, and throughput to area ratio were generated for all of these algorithms. Synthesis was performed using Synopsys Design Compiler, and placing & routing using Cadence Encounter Digital Implementation. All cores have been demonstrated to operate at the experimental clock frequencies from 15% to 92% higher then the worst case estimates returned by the placing & routing tools.

In [37], the throughput and power results, from the experimental evaluation of SHA-3 finalists, using 130nm ASIC technology, are reported. SASEBO-GII FPGA board is used as a controller. The obtained results indicated that the measured throughput was always lower than the Post-layout results, with the difference less than 30%.

In [38], a comprehensive evaluation of all Round 2 SHA-3 candidates, based on post-layout reports, using the 90 nm ASIC technology, is performed. The post-layout results were reported for two target throughputs, 20 Gbps and 0.2 Gbps, and the corresponding results compared against each other. No experimental measurements were performed as a part of this work. Similarly, in [21], post-layout throughputs are reported for all Round 2 candidates, using the ASIC 180 nm technology, and no experimental measurements are reported.

4.2 Design & Verification

4.2.1 System Design

The Zynq-7020 All-Programmable System on Chip (SoC) has been selected as our target device, due to its high performance and flexibility. In particular, our testbed takes advantage of software/hardware co-design and the memory management provided by the ARM platform. For each algorithm, the optimized C implementation [18] is first run on the ARM core and its performance recorded. As a result, after the conclusion of the hardware measurements, hardware vs. software speed up can be easily and fairly evaluated. Our system, composed of three major parts, is shown in Fig. 4.1:

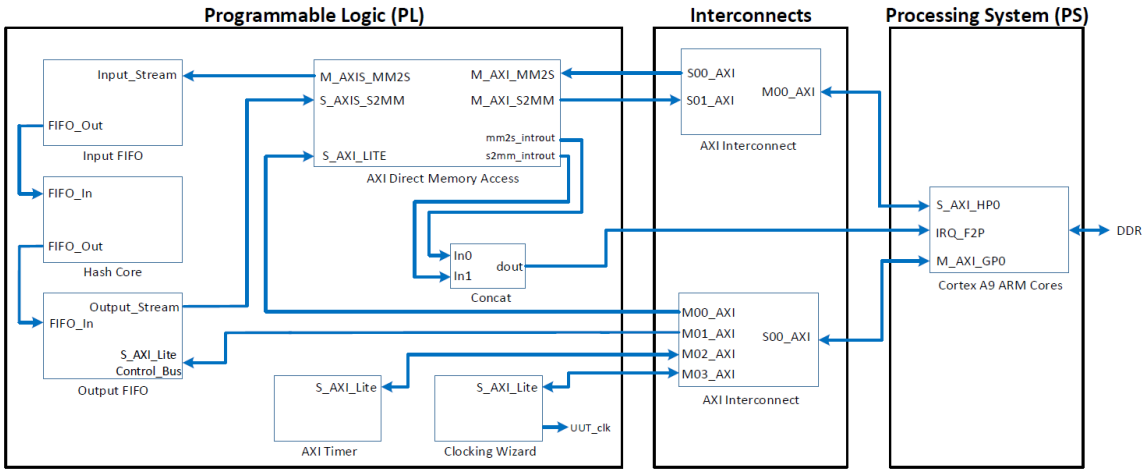


Figure 4.1: Block Diagram of the Testbed with the division into Programmable logic (PL), Interconnects, and Processing System (PS)

Processing System (PS)

The Processing System contains two Cortex A9 ARM Processor cores, and related logic. The HP (High Performance) ports are used for communication between PS and Programmable logic (PL). The input data (message) is sent to the hash core, located in PL. After the calculations are completed, an interrupt is generated, and the hash value is transferred back to the ARM core.

Interconnects

Two AXI Interconnect IPs take care of the data transmission between PS and PL using the memory mapped mode, AXI Full. These IPs are added and configured automatically using Vivado Design Suite.

Programmable Logic (PL)

Programmable Logic is used to implement the following major submodules:

1) Input FIFO: Input FIFO can be written to using AXI Stream Slave (AXIS) interface and read from using FWFT FIFO (First Word Fall Through FIFO) interface. It supports independent read and write clock domains. It was designed specifically for the purpose of this project, and its interface is shown in Fig. 4.2.

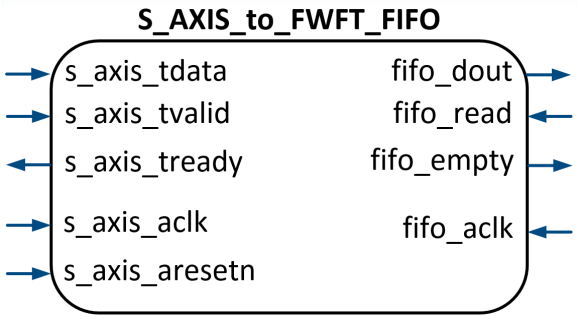


Figure 4.2: Input FIFO

2) Hash Core: Hardware implementation of a hash algorithm, compliant with the top-level interface and communication protocol described in [4]. This module can be replaced with any other hardware accelerator that can communicate with the FWFT FIFO interface.

3) Output FIFO: Output FIFO can be written to using the FWFT FIFO interface and read from using AXIS interface. It is also capable of handling independent clock domains for reading and writing. Moreover, it has an AXI Lite interface for configuring the transfer length and start delay information. Transfer length indicates the number of output words to be sent using the AXI Stream interface. Start delay lets the user to specify the delay, in clock cycles, before this module starts transferring the output data back to the processor. Delay countdown starts when the output is ready to send. Output FIFO was designed specifically for the purpose of this project, and its interface is shown in Fig. 4.3.

4) Clocking Wizard: Clocking Wizard was added from the Xilinx IP catalog [39] and is capable of generating a clock signal with a variable frequency, configurable from software.

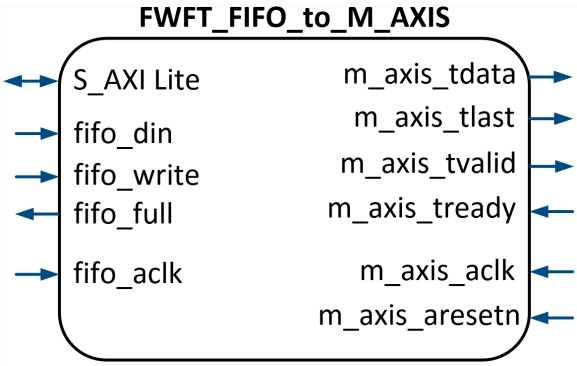


Figure 4.3: Output FIFO

This module can be controlled using the AXI Lite interface and let us change the clock frequency on the fly.

5) AXI Direct Memory Access (DMA): AXI DMA was added to block design from the Xilinx IP catalog [39], and converts the stream transaction protocol to the memory mapped protocol. As a result, it allows the hardware accelerator to read from and write to the DDR memory. The operation of this module is fully configurable from software, and frees the ARM processor to perform other tasks.

6) AXI Timer: AXI Timer is also a standard unit, available in the Xilinx IP catalog. It is capable of performing the execution time measurements for software and hardware implementations of various functions, with the accuracy of a single clock cycle of a system clock (by default: 10 ns).

7) Concat: The Concat module is used to concatenate two input signals and produce a single output, active when any of the two inputs is active. In the circuit from Fig. 4.1, it is used to create an interrupt to PS, active when either an input transfer or an output transfer is completed by AXI DMA.

In case a single clock domain was used, the maximum clock frequency supported by

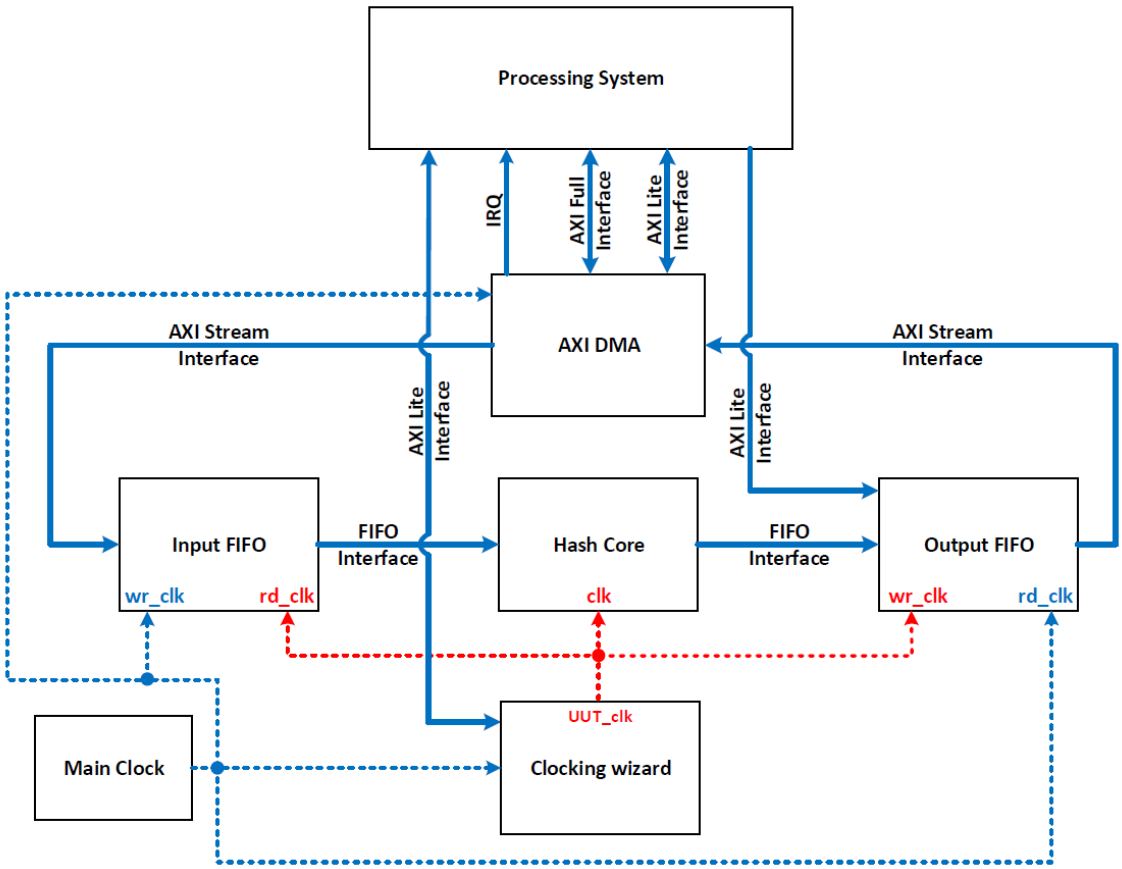


Figure 4.4: Simplified block diagram of the PL side with the indication of two independent clocks

our testbed would be limited by the maximum clock frequency of the AXI DMA and other standard IP components. To overcome this limitation, our testbed supports two separate clock frequencies, one for communication and auxiliary modules (AXI DMA, AXI Timer, etc.) and the other for our dedicated hash core. The system clock frequency, used for communication between PL and PS, is fixed. At the same time, the Clocking Wizard is used to generate the second clock, with a variable clock frequency, set on the fly, under the software control. Multiple frequencies of the second clock are then used during the binary search for the maximum frequency supported by a given hash core.

A simplified diagram of our testbed, with two independent clocks, is shown in Fig. 4.4.

In this testbed, at first AXI DMA, AXI Timer, and Interrupts are initialized. Then, Output FIFO is configured with the desired transfer length and start delay, and Clocking Wizard is configured with the UUT (Unit Under Test) output clock frequency. All initializations and configurations are done through the AXI Lite interface. Afterward, a buffer is allocated in DDR memory to store input data (message) of a certain size. Software implementation of the corresponding hash algorithm is run on ARM core. The software execution time is measured using AXI Timer. AXI Timer is started before the hash algorithm function call. Next, the message is transferred from the DDR memory through AXI DMA to Input FIFO. The Hash Core starts reading data from Input FIFO and writes the corresponding hash value to Output FIFO. Then, Output FIFO sends back the result to another buffer in the DDR memory through its AXI-Stream interface and AXI DMA. After the end of this transfer, the AXI Timer is stopped.

The entire end-to-end data transfer time and the hardware execution time are measured together using AXI Timer. The hash value received from the hardware side is compared with the value calculated by software. If they are equal, we increase the frequency of Hash core using Clocking Wizard, and rerun the entire process again. Otherwise, we decrease the frequency. The aforementioned procedure is repeated multiple times using the rules of binary search. The process ends only when we find the maximum clock frequency achievable by each hash algorithm with the precision of 0.1 MHz. The maximum experimental clock frequency, software execution time, hardware execution time at the maximum clock frequency, and the speed up (hardware vs. software) is reported.

4.2.2 Verification methodology

A universal testbench has been developed in the Vivado environment to verify the operation of our testbed using simulation. Apart from the circuit under test (composed of the Input FIFO, Hash Core, and Output FIFO), this testbench includes three AXI Traffic Generators (ATG) and one FIFO to simulate the functionality of the Zynq PS and AXI DMA. A simplified block diagram of the testbench is shown in Fig. 4.5. The first ATG module is

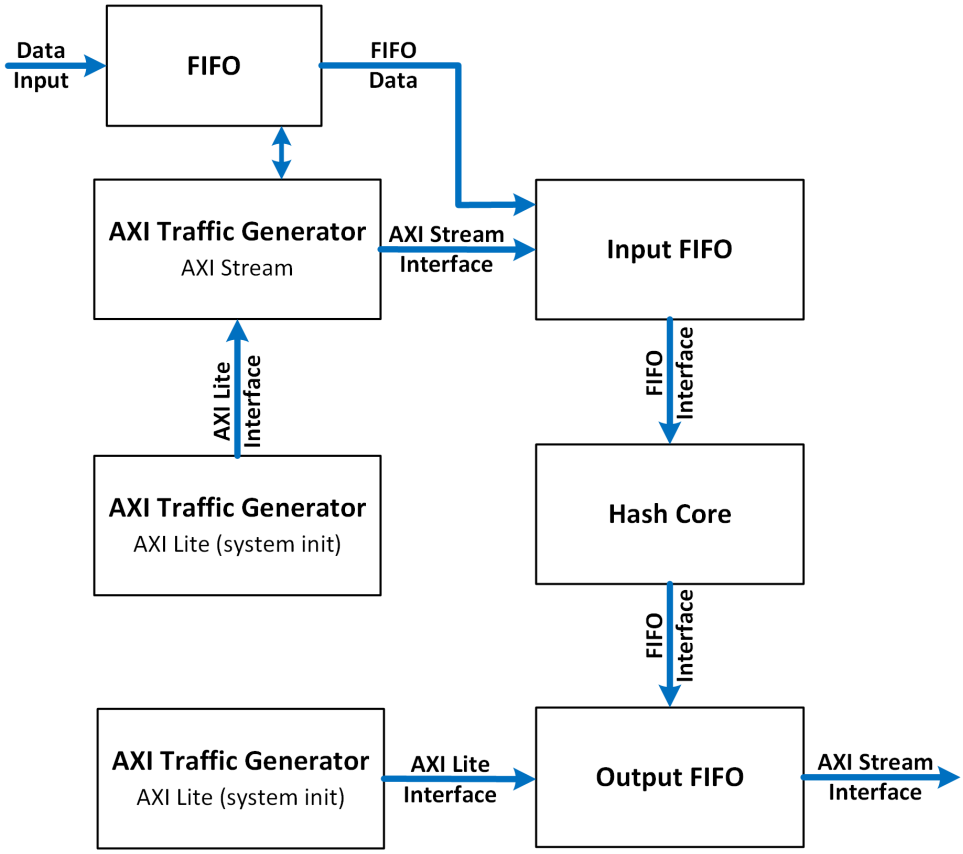


Figure 4.5: Universal testbench for Vivado environment

used in the AXI Stream mode to provide the control signals of the AXI Stream Interface to the Input FIFO module. Data that is provided by this ATG, configured in the AXI Stream mode, is random data. Therefore, an additional FIFO is included in the testbench to provide a source of desired input data. The second ATG module configures the first ATG module (AXI Stream ATG) with the desired configuration data, such as length of transaction, programmable delay, and the number of transactions, through AXI Lite interface. The third ATG module, configured in the AXI Lite mode, is used to configure Output FIFO. All AXI Traffic generators are preloaded with appropriate configurations, using the configuration COE files (address and data). At first the FIFO is filled with the input data. Then, all AXI Traffic Generators are started. The AXI Stream ATG and FIFO provide the input

message through the AXI Stream interface to the Input FIFO module. Hash Core reads data from Input FIFO, calculates hash value, and transfers it to the Output FIFO module. Eventually, we can compare the received hash value with the expected result to verify the functionality of the design.

4.3 Result

ZedBoard and Vivado 2015.4 have been used for result generation. All options of Vivado design suite including synthesis and implementation settings are set to default mode. On the software side, the bare metal environment and Xilinx SDK are used for running the C code on the ARM core of Zynq. The frequency of the primary system clock, connected to the interface IPs, including AXI DMA, AXI Timer, etc., is set to 100 MHz. Clocking Wizard generates the second clock, under the control of the C program, based on binary search. The frequency of Dual Core ARM (PS) is set to 667 MHz.

4.3.1 Maximum Frequency

Fig. 4.6 illustrates the maximum clock frequency achieved using static timing analysis and experimental testing, respectively, for each of the investigated algorithms. For all algorithms the experimental clock frequency is higher than that returned by static timing analysis. This result is expected, as CAD Tools alway take into account the worst case scenario, and thus report the pessimistic estimates in terms of speed. In particular, during any particular test, the critical path is not always triggered, even for a relatively long input. Additionally, a particular device used for testing (i.e., a particular instance of Zynq-7020 in our case) tends to have average rather than worst-case timing characteristics.

At the same our analysis reveals significant differences among the behavior of various algorithms. BLAKE, CubeHash, SHAvite-3, and Skein have an experimental frequency higher than the post-place & route frequency by 80 to 100%. In the second group, Fugue, JH, and Luffa, have the frequency higher by 55 to 65%. The third group includes Grøstl, Hamsi, and Keccak, and is characterized by the frequency improvement factor between 19

and 30%. Finally, ECHO and Shabal have almost identical frequencies returned by the static timing analyzer and the experimental test.

The algorithms belonging to the same group seem to have little in common in terms of basic operations, area requirements, or absolute value of the post-place and route frequency. As a result, the most likely explanation seems to be the placement of the respective designs in different locations on the chip, affected to a different extant by parameter variations.

Table 4.1 shows detailed values of the maximum clock frequency obtained from static timing analysis and experimental testing. Maximum experimental clock frequency was determined as a worst case value across all investigated input sizes from 10 to 5000 kB. The fourth column, contains the Throughput based on the formulas for the Throughput of each algorithm, listed in Table 4.2, with T replaced by the inverse of the Maximum Experimental Clock Frequency. The fifth column is the Throughput obtained by dividing the message input size by the actual execution time of hashing in hardware, measured using AXI Timer for the input size equal to 1000 kB.

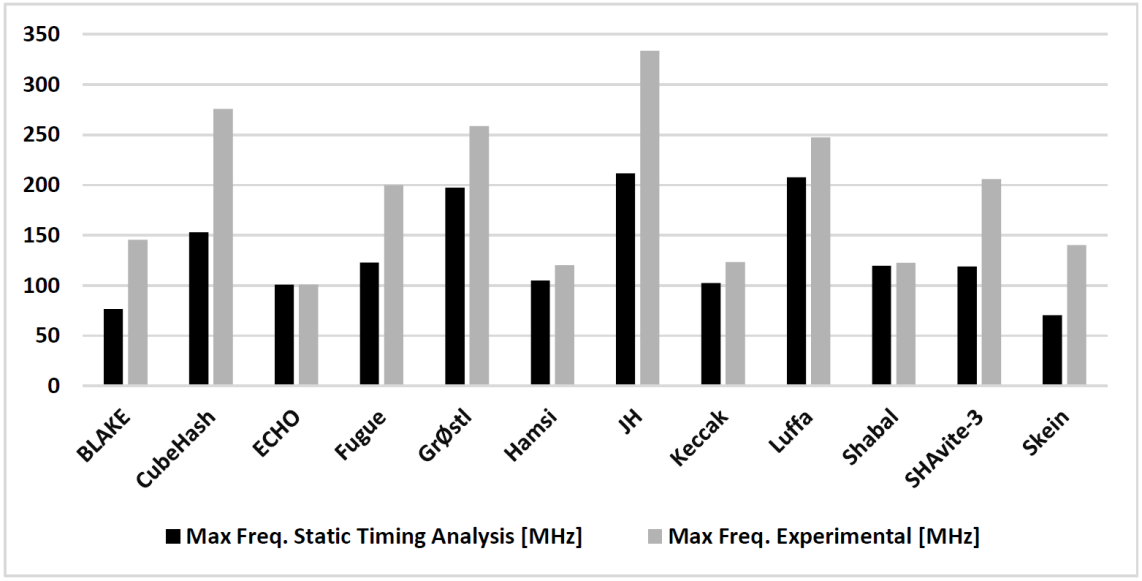


Figure 4.6: Maximum clock frequencies obtained using static timing analysis and the experimental measurement, respectively

Table 4.1: Detailed results for Maximum Frequencies and Throughputs

Algorithm	Max Freq. Static Timing Analysis [MHz]	Max Freq. Experimental [MHz]	Throughput Based on Formula and Max Exp. Freq. [Gb/s]	Throughput Based on Exp. HW Exe. Time [Gb/s]
BLAKE	76.4	145.4	3.546	3.544
CubeHash	152.9	275.8	4.413	4.399
ECHO	100.1	101.1	5.999	6.000
Fugue	122.9	200.0	3.200	3.191
Grøstl	197.2	258.6	6.305	5.821
Hamsi	105.0	124.9	1.333	1.332
JH	211.6	333.3	4.740	4.726
Keccak	102.6	123.1	5.292	5.314
Luffa	152.5	247.4	7.037	7.213
Shabal	119.7	122.5	0.981	0.983
SHAvite-3	119.0	205.7	2.846	2.828
Skein	70.6	140.3	3.782	3.772

4.3.2 Data transaction overhead

Fig. 4.7(a) shows the ratio of the hardware execution time measured using AXI Timer (including any communication overhead) over the calculated hardware execution time (using formulas provided in [4] and in Table 4.2), determined for different input message sizes. As we can see, for majority of investigated algorithms, such as BLAKE, CubeHash, ECHO, etc., the trend is very similar. The overhead of the communication between the PS and PL is between 20 and 50% for the messages of the size of 10 kB. For Shabal the respective value is about 5%. This overhead decreases below 5% for messages greater or equal to 100 kB. For 500 kB and larger messages, the relative overhead is very small and the ratio is almost 1. Luffa is the only exception to this trend. For this algorithm, the ratio is 2.2 for 10 kB of data and it decreases to 1.6 for 100 kB messages. For 500 kB and larger inputs, the ratio is stable around 1.55. Thus, the overhead is very substantial. The reason for this

unusual behavior is that Luffa is the only algorithm which has the Hash Core throughput exceeding the throughput of AXI DMA (6.4 Gbit/s at 100 MHz clock frequency). As a result, DMA core fails to feed the Hash Core with enough data to maintain the maximum possible throughput. Since the maximum clock frequency supported by the DMA Core on Zynq-7020 is 150 MHz [39], we can increase the DMA frequency to overcome this issue. Fig. 4.7(b) depicts the updated graph with new result for Luffa obtained using AXI DMA running at 150 MHz instead of 100 MHz. As illustrated by the graph, the DMA throughput bottleneck was completely eliminated and the behavior of Luffa is similar to that of other algorithms.

In addition, we repeated the same experiment using a second ZedBoard in order to demonstrate the effect of manufactoring variations on the maximum experimental clock frequency (and thus all other experimental timing measurements). In Table 4.3, we list the maximum frequency achieved using the first and second board, average maximum frequency, and standard deviation. It can be observed that the second board has slightly better results than the first one for the majority of algorithms and the same results for the remaining ones.

4.3.3 Hardware/Software execution time speed up

Table 4.4 shows HW/SW speed up for 5 different input sizes and all evaluated algorithms. As we can see, BLAKE and Shabal demonstrate speed up below 100. Hamsi, Luffa, Skein, ECHO, SHAvite-3 and Fugue achieve the speed-up from 100 to about 620. For CubeHash, Keccak and JH the speed-up exceeds 2,000, and for JH it is over 20,000. All of the aforementioned algorithms have almost stable results for messages larger than 100 kB. For 10 kB, the majority of them show the decrease in the speed-up caused by the communication overhead between software and hardware.

Table 4.2: The I/O Data Bus Width (in bits), Hash Function Execution Time (in clock cycles), and Throughput (in Gbits/s) for the 256-bit variant of SHA-3 candidates. T denotes the clock period in ns and N indicates the number of input blocks.

Algorithm	**I/O Bus width**	**Hash Time [cycles]**	**Throughput [Gbit/s]**
BLAKE	64	$2 + 8 + 21 \cdot N + 4$	$512/(21 \cdot T)$
CubeHash	64	$2 + 4 + 16 \cdot N + 160 + 4$	$256/(16 \cdot T)$
ECHO	64	$3 + 24 + 26 \cdot N + 1 + 4$	$1536/(26 \cdot T)$
Fugue	32	$2 + 2 \cdot N + 37 + 8$	$32/(2 \cdot T)$
Grøstl	64	$3 + 21 \cdot N + 4$	$512/(21 \cdot T)$
Hamsi	32	$3 + 1 + 3 \cdot (N - 1) + 6 + 8$	$32/(3 \cdot T)$
JH	64	$3 + 8 + 36 \cdot N + 4$	$512/(36 \cdot T)$
Keccak	64	$3 + 17 + 24 \cdot N + 4$	$1088/(24 \cdot T)$
Luffa	64	$3 + 4 + 9 \cdot N + 9 + 1 + 4$	$256/(9 \cdot T)$
Shabal	64	$2 + 64 \cdot N + 64 \cdot 3 + 16$	$512/(64 \cdot T)$
SHAvite-3	64	$3 + 8 + 37 \cdot N + 4$	$512/(37 \cdot T)$
Skein	64	$2 + 8 + 19 \cdot N + 4$	$512/(19 \cdot T)$

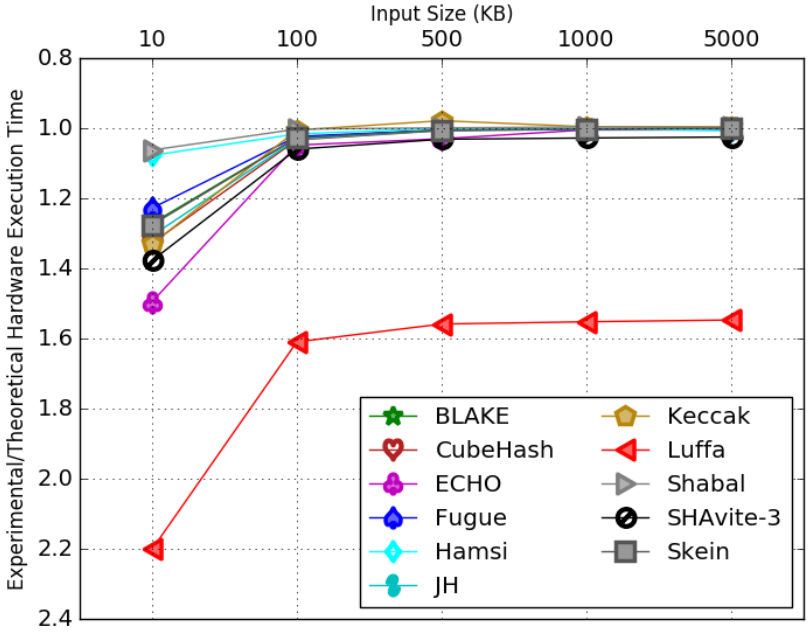
4.4 Conclusions

We have developed a novel experimental testbed, based on Zynq All Programmable System on Chip, for evaluating hardware performance of cryptographic algorithms competing in cryptographic contests, such as SHA-3, CAESAR, etc. This testbed allows determining the maximum experimental clock frequency of each core, using binary search, with the accuracy of 0.1 MHz. The operation of each hash core and surrounding FIFOs, can be first verified through simulation, and then tested experimentally using ZedBoard. The testbed can be used to correctly measure performance of designs with the maximum throughput of 64 bit $\cdot$ 150 MHz = 9.6 Gbit/s.

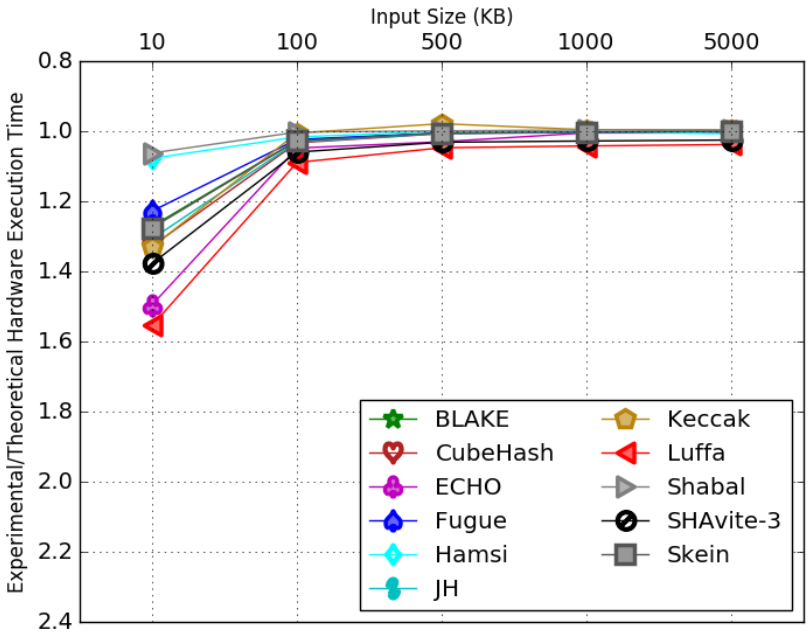
The correct operation of the testbed was demonstrated using the implementations of 12 Round 2 SHA-3 Candidates. For all these hash functions, the overhead of the communication between PS and PL was below 5% for 100 kB messages and negligible for messages above 500 kB. All algorithms have also demonstrated significant speed up vs. their execution in software on the same chip, in spite of the substantial speed of the ARM core, operating

at 667 MHz. Our experiments have also demonstrated that the maximum experimental clock frequency was always higher than the post-place and layout frequency calculated by Vivado, using static timing analysis. This fact demonstrates that the tool correctly returns the worst-case boundries, not likely to be reached in practice. At the same time, somewhat unexpectedly, the spread of ratios experimental to post-place and route frequency is very large, ranging from 1 to 2. This fact can be explained by a different influence of parameter variations, on the critical path of the each hash core, due to a different physical location (placement) of these critical paths in the FPGA fabric.

Our future work will involve an attempt at further explanation of the observed differences among various algorithms. We will also extend our environment to handle other types of cryptographic transformations, such as authenticated ciphers and post-quantum public key cryptosystems. Finally, we will also investigate at the use of other types of prototyping boards, including the FPGA boards with the PCI Express interface.



(a) DMA core running at 100 MHz for all algorithms



(b) DMA core running at 150 MHz in case of Luffa and 100 MHz for all other algorithms

Figure 4.7: Ratio of the Experimental (Measured) Hardware Execution Time to the Theoretical (Calculated) Hardware Execution Time for different input sizes (expressed in kilobytes)

Table 4.3: Experimental Maximum Frequency results on two different ZedBoards

Algorithm	Max Freq. Exp. Board1 [MHz]	Max Freq. Exp. Board2 [MHz]	Avr. Max Freq. Exp. [MHz]	Std. Dev. Max Freq. Exp. [MHz]
BLAKE	145.4	153.8	149.6	6
CubeHash	275.8	296.3	286.0	14
ECHO	101.1	101.1	101.1	0
Fugue	200.0	200.0	200	0
Grøstl	258.6	258.6	258.6	0
Hamsi	120.2	124.2	122.2	3
JH	333.3	347.8	340.5	10
Keccak	123.1	123.1	123.1	0
Luffa	247.4	262.5	254.9	11
Shabal	122.5	140.0	131.25	12
SHAvite-3	205.7	218.7	212.2	9
Skein	140.3	148.1	144.2	6

Table 4.4: HW/SW Speed Up for 5 Different Input sizes in kB

Input Size (KB)	**10**	**100**	**500**	**1000**	**5000**
BLAKE	73	87	88	89	89
CubeHash	3,326	4,105	4,165	4,181	4,184
ECHO	287	384	397	399	400
Fugue	127	119	119	119	119
Hamsi	581	619	624	624	625
JH	19,155	24,109	24,639	24,712	24,776
Keccak	2,341	3,079	3,169	3,182	3,192
Luffa	302	398	410	411	412
Shabal	6	6	6	6	6
SHAvite-3	81	102	104	105	105
Skein	92	112	114	115	115

Chapter 5: Conclusion

High-level synthesis (HLS) offers a potential to allow hardware evaluation at the early stages of cryptographic contests, such as SHA-3 and CAESAR, when the number of candidates is still very large (e.g., greater than 50 in Round 1 of both aforementioned contests). HLS-based design significantly reduces hardware development time, and can be carried out by a cryptographer with limited experience in hardware design. The designer still needs to modify the reference code in C or C++ to meet the requirements of Vivado HLS tool and improve the efficiency of the generated HDL code. Quite often, developing an efficient HLS-ready C code requires some basic knowledge of hardware design and target hardware architecture. The programmer needs to rewrite and tweak the HLS-ready C code and then repeatedly check the estimated results, to finally achieve the desired hardware architecture. This check can be done easily by determining the number of clock cycles per input block, which should be close to the expected value, determined by a cryptographic algorithm and its hardware architecture (e.g., equal to the number of cipher rounds plus a small constant, such as 1 or 2, for the basic iterative architecture). Our results for four Round 2 SHA-3 candidates, obtained using the RTL and HLS based methodologies, respectively, have demonstrated quite substantial correlation in terms of rankings according to the three major hardware performance metrics: throughput, area, and throughput to area ratio.

After the RTL code is generated using either the RTL or HLS based approach, this code must be processed by FPGA tools, such as Xilinx Vivado, in order to produce the best possible results after logic synthesis, mapping, placing, and routing. The results are affected by multiple tool options as well as the requested clock frequency. For older Xilinx FPGAs (up to Virtex 6 and Spartan 6) Xilinx ISE was a primary integrated development environment used for processing of HDL code. Starting from the Xilinx Series 7 FPGAs (Virtex 7, Artix 7, and Kintex 7) the primary tool of choice, expected to produce better

results in a shorter amount of time, is Xilinx Vivado. The option optimization tool developed by the GMU group in the period 2010-2014, called ATHENa (Automated Tool for Hardware Evaluation), is capable of working only with Xilinx ISE. Therefore, in this project we have made the first attempt at replacing ATHENa, for the purpose of generating optimized results using Xilinx Vivado, by a new tool, we called Minerva. Minerva searches for the best requested clock frequency and the best set of tool options, leading to the highest achieved clock frequency, or the highest achieved frequency to area ratio, after static timing analysis. In addition, Minerva takes advantage of multithreading and multi core execution to reduce the run time. It can apply an arbitrary number of preselected tool option sets (called optimization strategies) and combine it with the frequency search in order to achieve the best results in terms of the throughput or throughput to area ratio. The results for 21 Round 2 CAESAR candidates indicate that we can achieve up to 37% improvement in terms of the throughput to area ratio in comparison to a simpler binary search for optimal requested clock frequency, with default values of all tool options. The average run time depends mostly on n (number of runs in parallel) which was 16 in our experiments. This average run time is almost 6 times larger than in case of binary search with default values of parameters.

After the optimized FPGA bitstream is produced, this bitstream can be tested experimentally, using prototyping boards. In this project, we have decided to use ZedBoard based on Xilinx Zynq All Programmable System on Chip. Our experimental testbed and the associated software employs binary search to determine the maximum experimental clock frequency of each cryptographic core. The maximum supported throughput, determined by the maximum throughput of the AXI DMA, used for communication between the Processing System (based on ARM Cortex A9) and Programmable Logic (including a hardware cryptographic unit) of Zynq is 9.6 Gbit/s. Twelve SHA-3 Round 2 candidates were investigated in this work from the point of view of the maximum experimental clock frequency and the maximum experimental throughput. The obtained results indicate that the communication overhead is negligible for messages above 500 kB. Moreover, all algorithms

achieved significant speed up vs. their execution in software and maximum experimental frequency was always greater than the value reported by Xilinx Vivado after static timing analysis.

In summary, the aforementioned HLS development methodology, option optimization with Minerva, and the experimental setup based on Zynq provide an efficient, reliable and comprehensive package for accelerated development and benchmarking of a large number of algorithms competing with each other during cryptographic contests.

Bibliography

Bibliography

[1] National Institute of Standards and Technology. (2000, Oct) Report on the development of the advanced encryption standard (aes). [Online]. Available: http://csrc.nist.gov/archive/aes/round2/r2report.pdf

[2] ——. (2012, Nov) Third-round report of the sha-3 cryptographic hash algorithm competition. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf

[3] E. Homsirikamol and K. Gaj, *Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study*. Cham: Springer International Publishing, 2015, pp. 217–228. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_18

[4] 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.

[5] ARM. AMBA Specifications. [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications.php

[6] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[7] C. Maxfield. (2012, Jul) First public access release to xilinx vivado design suite. [Online]. Available: http://www.eetimes.com/document.asp?doc id=1317376

[8] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 1102–1105.

[9] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: Productivity, performance, and software constraints," *JECE*, vol. 2012, pp. 1:1–1:1, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/649057

[10] G. Quan, J. P. Davis, S. Devarkal, and D. A. Buell, "High-level synthesis for large bit-width multipliers on fpgas: A case study," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '05. New York, NY, USA: ACM, 2005, pp. 213–218. [Online]. Available: http://doi.acm.org/10.1145/1084834.1084890

[11] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. B. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," in *2007 3rd Southern Conference on Programmable Logic*, Feb 2007, pp. 99–106.

[12] F. Gruian and M. Westmijze, "Vhdl vs. bluespec system verilog: A case study on a java embedded architecture," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 1492–1497. [Online]. Available: http://doi.acm.org/10.1145/1363686.1364037

[13] I. Berkeley Design Technology. (2010) High-level synthesis tools for xilinx fpgas. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/Xilinx hlstcp.pdf

[14] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[15] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–8.

[16] 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 *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, ser. FPL '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 414–421. [Online]. Available: http://dx.doi.org/10.1109/FPL.2010.86

[17] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 6th ed., Prentice Hall, 2013.

[18] NIST. (2015, Aug) Cryptographic HASH and SHA-3 Standard Development. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/index.html Accessed Oct. 10, 2016

[19] R. Lien, T. Grembowski, and K. Gaj, *A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 324–338. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24660-2_25

[20] A. Akin, A. Aysu, O. C. Ulusel, and E. Savas, "Efficient hardware implementation of high throughput sha-3 candidates keccak, luasaa and blue midnight wish for single- and multi-message hashing. 2nd sha-3 candidate," in *SIN*, Russia, 2010.

[21] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, "High-speed hardware implementations of blake, blue midnight wish, cubehash, echo, fugue, grøstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein," Cryptology ePrint Archive, Report 2009/510, 2009, http://eprint.iacr.org/2009/510.

[22] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "A compact fpga implementation of the sha-3 candidate echo," Cryptology ePrint Archive, Report 2010/364, 2010, http://eprint.iacr.org/2010/364.

[23] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive evaluation of high-speed and medium-speed implementations of five sha-3 finalists using xilinx and altera fpgas," Cryptology ePrint Archive, Report 2012/368, 2012, http://eprint.iacr.org/2012/368.

[24] George Mason University. (2016, Oct) GMU source code. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id= source codes

[25] ——. Cerg. gmu athena database. [Online]. Available: https://cryptography.gmu.edu/athena

[26] K. Gaj, E. Homsirikamol, and M. Rogawski, *Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 264–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15031-9_18

[27] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "Gmu hardware api for authenticated ciphers," Cryptology ePrint Archive, Report 2015/669, 2015, http://eprint.iacr.org/2015/669.

[28] M. Goosman, R. Shortt, D. Knol, and B. Jackson, "Exploreahead extends the planahead performance advantage," in *Xcell Journal*, Third Quarter 2006, pp. 62–64.

[29] ebacs: Ecrypt benchmarking of cryptographic systems. [Online]. Available: http://bench.cr.yp.to

[30] xilinx. Vivado design suite user guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug904-vivado-implementation.pdf

[31] D. Fedoryka, "Fast Implementation of the Secure Hash Algorithm SHA-256 in FPGA," Master's thesis, George Mason University, July 2004.

[32] K. Kobayashi, J. Ikegami, M. Knežević, X. Guo, S. Matsuo, S. Huang, L.Nazhandali, U. Kocabas, J. Fan, A. Satoh, I. Verbauwhede, K. Sakiyama, and K. Ohta, "A Prototyping Platform for Performance Evaluation of SHA-3 Candidates," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2010)*. IEEE, Jun 2010, pp. 60–63.

[33] M. Knežević, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, U. Kocabaş, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki, "Fair and consistent hardware evaluation of fourteen round two sha-3 candidates," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 5, pp. 827–840, May 2012. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2011.2128353

[34] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "Fpga implementations of the round two sha-3 candidates," in *2010 International Conference on Field Programmable Logic and Applications*, Aug 2010, pp. 400–407.

[35] ——, "A hardware wrapper for the sha-3 hash algorithms," Cryptology ePrint Archive, Report 2010/124, 2010, http://eprint.iacr.org/2010/124.

[36] F. K. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J. P. Kaps, "Lessons Learned from Designing a 65 nm ASIC for Evaluating Third Round SHA-3 Candidates," in *The Third SHA-3 Candidate Conference*, Washington DC, 2012.

[37] X. Guo, M. Srivastav, S. Huang, D. Ganta, M. B. Henry, L. Nazhandali, and P. Schaumont, "ASIC implementations of five SHA-3 finalists," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar 2012, pp. 1006–1011.

[38] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. K. Gürkaynak, *Developing a Hardware Evaluation Method for SHA-3 Candidates*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 248–263.

[39] I. Xilinx. (2015) AXI DMA v7.1 LogiCORE IP Product Guide (PG021).

Curriculum Vitae

Farnoud Farahmand graduated from Islamic Azad University, Karaj Branch with a Bachelor of Science degree in Electrical Engineering in 2013. During his time in graduate school, he has been involved in teaching various graduate and undergraduate courses at George Mason University. In addition, he worked as a research assistant in GMU Cryptographic Engineering Research Group (CERG). He specializes in prototyping and hardware benchmarking of various algorithms. His notable work includes, but is not limited to, HLS-based implementation of Round-2 SHA-3 candidates, the development of an automated tool for hardware evaluation (Minerva), and design and development of a Zynq-based Testbed for the experimental benchmarking of algorithms competing in cryptographic contests. His research efforts during graduate school have led to the publication of four noteworthy papers in the field of computer engineering.