Accelerating RSA Public Key Cryptography via Hardware Acceleration

Item Type	Thesis (Open Access)
Authors	Ramesh, Pavithra
DOI	10.7275/15914936
Download date	2025-03-23 10:20:01
Link to Item	https://hdl.handle.net/20.500.14394/33950

ACCELERATING RSA PUBLIC KEY CRYPTOGRAPHY VIA HARDWARE ACCELERATION

A Thesis Presented

by

PAVITHRA RAMESH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2020

Electrical and Computer Engineering

ACCELERATING RSA PUBLIC KEY CRYPTOGRAPHY VIA HARDWARE ACCELERATION

A Thesis Presented

by

PAVITHRA RAMESH

Approved as to style and content by:

Sandip Kundu, Chair

Russell Tessier, Member

Wayne Burleson, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Prof. Sandip Kundu, my thesis and graduate advisor for guiding me throughout this work, for consistently finding time for me in his busy schedule, for being patient with me, for his immense knowledge and for his valuable suggestions at all stages. Without his persistent help, taking this thesis to completion would not have been possible. His contagious enthusiasm was my primary source of inspiration and is something that I would hold onto throughout my life.

I would also like to thank my thesis committee members, Prof. Russell Tessier and Prof. Wayne Burleson for being part of my committee, for taking the time to evaluate the work and for providing suggestions and constructive comments.

My parents, Ramesh Chandran, Geetha Ramesh and my husband, Sriraam, have been my cheerleaders throughout my period of study. No mere expression of gratitude would suffice to thank them for their patience and unwavering support.

I would also like to thank Dr. Saravana Manickam, for helping me familiarize with the research process and for his guidance throughout.

Last, but not the least, I would like to thank my friends Mythili Vishalini Anbazhagan, Deepa Rajam Viswanathan, Harikrishnan S Pillai, Rahul Raj and Naveen Kumar Dumpala for lending a ear whenever I needed it, during this period of study.

ABSTRACT

ACCELERATING RSA PUBLIC KEY CRYPTOGRAPHY VIA HARDWARE ACCELERATION

FEBRUARY 2020

PAVITHRA RAMESH

B.Tech., MODEL ENGINEERING COLLEGE, KOCHI, KERALA, INDIA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sandip Kundu

A large number and a variety of sensors and actuators, also known as edge devices of the Internet of Things, belonging to various industries - health care monitoring, home automation, industrial automation, have become prevalent in today's world. These edge devices need to communicate data collected to the central system occasionally and often in burst mode which is then used for monitoring and control purposes. To ensure secure connections, Asymmetric or Public Key Cryptography (PKC) schemes are used in combination with Symmetric Cryptography schemes. RSA (Rivest - Shamir- Adleman) is one of the most prevalent public key cryptosystems, and has computationally intensive operations which might have a high latency when implemented in resource constrained environments. The objective of this thesis is to design an accelerator capable of increasing the speed of execution of the RSA algorithm in such resource constrained environments. The bottleneck of the algorithm

is determined by analyzing the performance of the algorithm in various platforms - Intel Linux Machine, Raspberry Pi, Nios soft core processor. In designing the accelerator to speedup bottleneck function, we realize that the accelerator architecture will need to be changed according to the resources available to the accelerator. We use high level synthesis tools to explore the design space of the accelerator by taking into consideration system level aspects like the number of ports available to transfer inputs to the accelerator, the word size of the processor, etc. We also propose a new accelerator architecture for the bottleneck function and the algorithm it implements and compare the area and latency requirements of it with other designs obtained from design space exploration. The functionality of the design proposed is verified and prototyped in Zynq SoC of Xilinx Zedboard.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

INTRODUCTION

Internet of Things (IoT) is considered to be the third wave of innovation in information technology, to follow the earlier waves of Internet and mobile computing. IoT devices enable useful services by integrating computation, communication, and sensing capabilities, gathering and processing data from which intelligence is extracted to enable services towards a smarter world. The number of IoT edge devices connected to the Internet already surpasses the world's population with economic impact predicted in trillions of dollars. The number of devices connected to the internet has gone from 500 million in 2003 to 12.5 billion in 2010 and is predicted to reach 50 billion by 2020 [5]. The advancements in IoT technology have allowed a variety of edge devices to penetrate into the real world like never before. IoT devices help monitor, control and streamline operations across sectors such as consumer, healthcare, industrial, transportation and military, to name a few. Consumer applications like smart home, elder care, medical and healthcare applications like remote health monitoring, emergency notification systems, transportation industrial applications like smart traffic control, smart parking and military applications for surveillance and combat related objectives are enabled due to sensors like temperature, pressure, accelerometer, motion detection, image, chemical, IR among others. Sensors and actuators, also known as edge devices are resource constrained and usually have microcontroller units or processors with limited computing and networking capabilities, to enable exchange of information to other entities on the network or to communicate with the central system. With ubiquity of IoT devices in all facets of our lives, privacy and security risks of the edge devices have emerged as a dominant concern as they continuously collect and share data about our surroundings. Ensuring security across

all levels of IoT architecture is essential, however it is more crucial to enable security measures at the end nodes. Such devices, if deployed in an unprotected environment, could become prey to attackers eavesdropping information, providing unauthorized access to device or data, allowing data to be tampered with, or even attach other unauthorized devices to the network. Thus, it has become an utmost necessity to implement cryptography to provide security, ensure privacy, authentication among other services in the end nodes which are resource constrained with low memory, and limited computational capabilities operating on a string power budget.

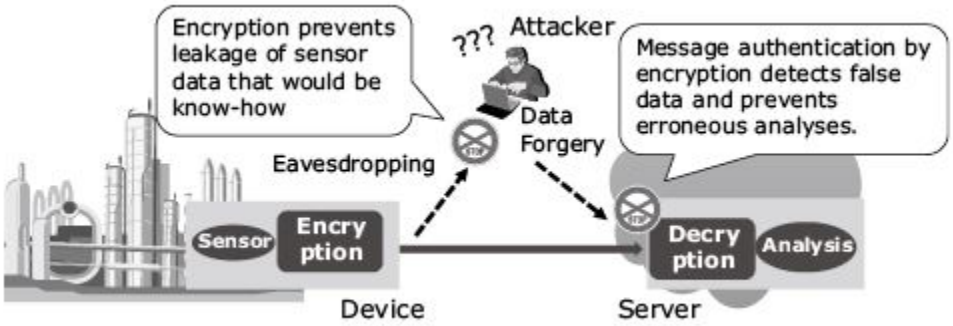

Figure I.1: Cryptographic schemes against attack on data collection

Many applications do not demand continuous connectivity. In the medical field, sensors could be used to collect medical data from a patient's body, but only communicate with the health care server system, if the data sensed is abnormal. Sensors could also be used for sensing voltage, humidity, temperature etc. which require the central system to be notified only if the measured data changes considerably that demands control actions to be taken. Some of the edge devices pack data into batches to minimize network power. If the nature of the application does not require continuous connection batched data transmission is the preferred option. Maintaining a continuous network connection is power consuming, also makes it easier for attackers to access the devices through network and hence is not preferred. Establishing a connection as and when needed in the form of sessions is much better suited for such

applications which transmit/receive data in bursts as shown in figure I.2. Each such session needs to follow an authentication protocol and data encryption protocols for ensuring secure transfer of data to valid entities. Public Key Cryptography (PKC) schemes are commonly used to implement key exchange and authentication protocols. These security schemes involve three processes: Key generation, encryption and decryption. Generating new set of keys for each session ensures that the keys do not have to be stored in the end nodes, preventing it from being tampered. Generating new set of keys per session also enables the network to be disconnected when not it use.

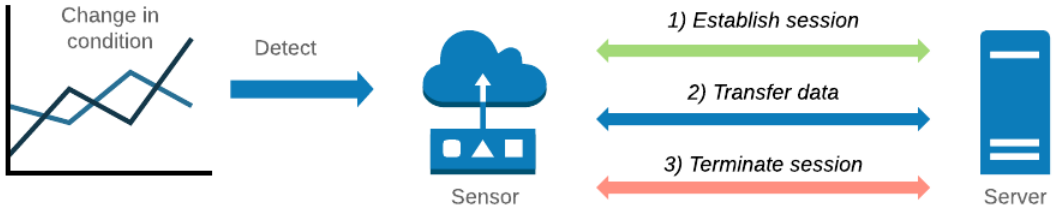

Figure I.2: Session based data transfer between sensors and servers

Implementing PKC schemes on such resource restricted environments are usually computationally expensive and time consuming. Such System on Chips (SoCs) would benefit from having a dedicated purpose-built circuitry, also known as an accelerator, that performs a portion of a task and helps in increasing the overall performance. Embedding hardware accelerators in SoC can also lead to energy efficiency improvement at the cost of a slightly larger die. Enabling hardware acceleration is a two step process - identification of the bottleneck and designing accelerator architecture for the bottleneck function. To ensure optimum performance gain, the hardware accelerator function has to be carefully chosen so that the most performance can be obtained from minimal hardware investment.

In this work, we consider the RSA algorithm [6], the most prevalent Public Key Cryptography scheme today and analyze it to determine the bottleneck function which is responsible for a substantial amount of overall execution time. We run the RSA algorithms on three platforms with varying processing power - an Intel Linux machine, Raspberry Pi 3 Model B+ running an ARM processor and a Nios II soft processor configured on an Altera FPGA and analyze the execution of the algorithm in all three platforms to determine the bottleneck of the algorithm.

We then aim to design a minimal accelerator, that is suitable for such resource restricted devices and capable of speeding up the bottleneck function and in effect the whole algorithm. Crypto-coprocessors and hardware accelerators for speeding up the RSA encryption and decryption are very popular, but these circuits take the entire algorithm implementation load off of the processor. We aim to design an accelerator that is minimal in functionality but is still capable of accelerating the overall algorithm.

We take into consideration system level aspects like the input availability for the accelerator which is a key factor in determining the overall latency of the circuit. According to the resources available, for instance, the number of ports, that help in transferring data to the accelerator, the design will need to be changed.

After determining the hotspot function, algorithmic variations are considered to determine the best possible design under restricted input availability. An architectural variation to implement the hotspot function is proposed. Using Vivado HLS, the latency and area requirements of the proposed design is compared with that of other architectures to implement the hotspot function under similar system constraints. We configure the Zynq SoC with our proposed accelerator design as IP and prototype in Zedboard to verify its functionality.

Chapter 1 discusses the context and background information relevant to the work. In chapter 2, the results of performance profiling of the RSA system on embedded

devices are discussed. Chapter 3 deals with design approach taken for accelerator design, the proposed accelerator architecture and the architectures taken for comparison. Chapter 4 discusses the testbench for experiments, experimental setup, observations and analysis of results obtained. Chapter 6 concludes the work and discusses possible extension to the work.

CHAPTER 1

PROBLEM STATEMENT AND CONTEXT

1.1 Cryptographic schemes for securing edge devices

Edge devices like sensors and actuators form the backbone of Internet of Things. At the same time, they are the weakest links posing a threat to both security and privacy. In August 2019, NIST has released a draft listing Security feature recommendations for IoT Devices [7]. The trend in enabling IoT security is shifting from being optional to being essential and mandatory.

Proper use of cryptographic schemes is essential to securing all edge devices. The cryptographic schemes can be classified into two categories, namely symmetric or private key cryptography and asymmetric or public key cryptography (PKC). Both help in achieving the following objectives or services [8]:

- *Confidentiality* : A service that ensures that information is accessible to only authorized entities.
- *Integrity* : A service that ensures that information and system modifications can be done only by authorized entities.
- *Authentication* : A service that helps in assuring that the origin of a message is correctly identified.
- *Non-repudiation* : A service that ensures that neither the sender nor the receiver of a transmission can deny its actions.

Symmetric key cryptography schemes are based on algorithms which use the same key for encryption and decryption or where the decryption key is easily derivable from the encryption key. These schemes are usually used for high speed data encryption. However, the security of the scheme is entirely based on the secrecy of the key. One also needs to ensure availability of means for the entities to agree on and exchange keys in a secure manner, also referred to as the key distribution problem.

Public Key cryptography or asymmetric cryptography schemes consists of a set of two keys - public key for encryption and private key for decryption. An encrypted message can be sent by anyone to entity A, using *A's* public key, but only A can decrypt the message using its private key. *A's* private key must be maintained secret. The security of the scheme lies in the fact that it is impossible for anyone except A to derive the private key from public key (at least in a reasonable amount of time). The public key cryptography algorithms can be generally divided into three categories - algorithms based on integer factorization problem (*RSA*), algorithms based on discrete logarithm problem (*Diffie-Hellman*), algorithms based on Elliptic curves (*ECC*). Regardless of the algorithm used, all of them perform complex and computationally intensive operations on very large numbers. The most common operation in PKC is modular exponentiation, i.e., the calculation of $x^e \ mod \ n$. Exponentiation with large numbers, as required by PKC, is extremely arithmetically expensive.

The key distribution problem of symmetric key systems can be solved by using asymmetric key schemes, but at the cost of expensive arithmetic computations. In practice, cryptographic systems are usually a mix of symmetric key and asymmetric key schemes. The former is used for encryption and data transfer, whereas the latter is preferred for key establishment and authentication through digital signatures [9]. The basic principle of digital signature works as follows. If B wants to authenticate A, A sends a message after encrypting with her secret private key. If the public key of A which is known to B is capable of decrypting the encrypted message sent by A,

B can authenticate that the message is from A. The validity of the public key of A is commonly verified using Public Key Infrastructure (PKI) via a Certificate Authority (CA). Certificate authorities issues digital certificates, which binds public key to the entities. When a CA issues a signed digital certificate to A, it allows B to trust the public key of A, since B trusts the CA.

Implementing a combination of symmetric and asymmetric schemes is essential to enable security in edge devices. In this work, we focus on the RSA algorithm, the most prevalent asymmetric cryptographic scheme and design of a suitable hardware accelerator enabling resource constrained devices to implement them with reasonable performance.

1.2 Relevance of RSA

Even though Elliptic Curve Cryptography (ECC) offers smaller key sizes and have better memory, energy and bandwidth savings, RSA is still the most prevalent public-key scheme in the Internet today. In many organizations, the RSA algorithm is used for employee verification. Employees have chip based smart cards, which usually run RSA in combination with some other algorithm to ensure security. One such product is IDPrime MD 3811 smart cards, a single chip based dual interface smart card by Gelmato, which is secured by both the RSA and ECC algorithms. It addresses a wide range of use cases that require PKI security, including secure acesss, email encryption, secure data storage, digital signatures and secure online transactions for end users and has the ability to run RSA-2048 and generate on-card asymmetric RSA keys [10]. Another such product is IDPrime PIV (Personal Identity Verification) card v1.55 by Gelmato which allows physical and logical access to Federal amenities for Federal employees and was published as FIPS PUB - 201. It has the ability to run RSA (1024-bit, 2048-bit) and on-card key pair generation [11].

In [12], an end to end security protocol based on the RSA algorithm was proposed for IoT devices. The authors have chosen RSA over ECC, since it is the dominant PKC system and hence suitable infrastructure for obtaining certificates from certificate authorities (CAs) are already in place. Also, in [13], it is mentioned that public key signature validation is faster in RSA compared to ECC.

1.3 RSA Algorithm

Before delving into accelerator design, we look at the RSA algorithm in detail. RSA (Rivest-Shamir-Adleman) is one of the commonly used public key cryptographic algorithms. The strength of the RSA algorithm lies in the difficulty of factoring large numbers. RSA involves three processes: *Key generation, Encryption, Decryption*. Most commonly, after the RSA keys are generated in a device, say A, the public key is distributed to the device, say B which wants to establish communication with A. Private key is stored in the device A. The storage of key is a major security concern. A more secure way is to generate the key as and when communication must to be established. In this scheme, with the accelerator, the device should be capable of supporting encryption, decryption and key generation.

1.3.1 RSA key generation

An RSA public key (n, e) consists of a modulus n and a public key exponent, e. The modulus n, is calculated as the product of two positive prime integers p and q. The size of an RSA key pair is considered to be the size of modulus n in bits. The RSA private key corresponding to the public key (n, e) is (n, d), where d is the private key exponent. The private key exponent d, depends on the public key exponent, e and the modulus n [14].

The RSA private key exponent d, the two positive prime integers p and q must be kept as a secret to ensure security. Revealing the public key exponent e and the modulus n does not compromise security and may be made known to anyone [14].

In NIST SP 800-57, it is specified that for use in RSA signature algorithms, the recommended lengths for n are 1024, 2048 and 3072 bits as it can be seen from table 1.1 [15]. The corresponding sizes of positive primes p and q are 512, 1024 and 1536 bits respectively. In this work, we consider key size to be 2048 and hence the corresponding positive prime sizes to be 1024.

Algorithm security lifetime	**RSA key size**
Through 2010	*Min: k=1024*
Through 2030	*Min: k=2048*
Beyond 2030	*Min: k=3072*

Table 1.1: Recommended minimum key size for RSA

According to the authors of the original paper [6], the following steps have to be followed for calculating keys:

1. Choose two random prime numbers p and q

2. Calculate modulus n, using

$$n = p \times q \tag{1.1}$$

3. Calculate private key exponent d from p and q such that it satisfies,

$$gcd(d, (p-1) \cdot (q-1)) = 1 \tag{1.2}$$

4. Calculate private key exponent e from p, q, d to be the multiplicative inverse of d, modulo*(p-1).(q-1)* i.e.,

$$e \cdot d \equiv 1 \cdot (mod(p-1) \cdot (q-1)) \tag{1.3}$$

1.3.2 RSA encryption

The *ciphertext*, C is calculated by the following function where P is the *plaintext*, *(n,e)* is the *public key*,

$$C = P^e \ mod \ n \tag{1.4}$$

1.3.3 RSA decryption

The *plaintext*, P is calculated by the following function where C is the *ciphertext*, *(n,d)* is the private key,

$$P = C^d \ mod \ n \tag{1.5}$$

1.4 RSA key generation in practice

Current implementations of RSA choose public key exponent, e and then compute private key exponent d from it. The FIPS 186-4 specifies that the public key exponent e can be selected with the following constraints [14]:

1. The public key exponent e can be selected prior to generating positive primes p, q and the private key exponent d

2. The public key exponent can be an odd positive integer in the range,

$$2^{16} < e < 2^{256}$$

3. The public key exponent can be either a fixed or random value that satisfies the constraint mentioned in 2.

After the selection of public key exponent, suitable primes p and q are selected with the following constraints:

1. The positive primes p and q can be selected in such a way that $(p-1)$ and $(q-1)$ are relatively prime to the public key exponent.

2. The primes p and q should be selected randomly and should satisfy conditions,

$$(\sqrt{2})(2^{(nlen/2)-1} \leq p \leq (2^{nlen/2} - 1)) \tag{1.6}$$

$$(\sqrt{2})(2^{(nlen/2)-1} \leq q \leq (2^{nlen/2} - 1)) \tag{1.7}$$

3. The primes p and q should be such that,

$$|p - q| > 2^{(nlen/2)-100} \tag{1.8}$$

The private key exponent d, is generated after the generation of primes p and q with the following constraints,

1. The following inequality holds for d,

$$2^{(nlen/2)} < d < LCM(p-1, q-1) \tag{1.9}$$

2. The value of d should be such that,

$$d = e^{-1} \; mod \; (LCM(p-1, q-1)) \tag{1.10}$$

It is convenient to use a small value of e for encryption purposes. 65537 is a popular choice for e. In this work, we choose public key exponent as 65537 and choose the primes p and q randomly such that the constraints mentioned above are satisfied. The private key exponent, d is chosen such that equation 1.10 is satisfied.

1.5 Miller-Rabin Primality test

In RSA key generation, we observe that after selecting public key exponent, suitable primes numbers are generated such that it satisfies criteria mentioned in equation

(1.6) and (1.7). Primality tests are usually used to generate prime numbers and they can be broadly classified into three categories - probabilistic, deterministic and heuristic. In probabilistic primality tests, a number is chosen randomly from a sample space and checked whether some equality involving it is true. If the equality holds, to a certain accuracy, the number is determined to be probably prime. Examples under this category include Fermat test, Solovay-Strassen test and Miller-Rabin test [16]. Deterministic primality tests are capable of determining a number to be prime with absolute certainty, but its running times are usually higher than probabilistic tests. For generating primes suitable for the RSA key generation purposes, it is sufficient to use Miller-Rabin test.

Algorithm 1 Miller-Rabin primality test [1]

Require: An odd integer $n > 2$ to be tested ; a positive integer t indicating number of trials

Ensure: n is prime or composite

1: Find v and w such that $n - 1 = 2^v \cdot w$
2: **for** $j \leftarrow 1$ *to* t **do**
3: Choose random a such that $0 < a < n$
4: Set $b \leftarrow a^w \ mod \ n$
5: **if** $b = 1$ or $n = 1$ **then**
6: go to step 18
7: **end if**
8: **for** $i \leftarrow 1$ *to* $v - 1$ **do**
9: Set $b \leftarrow b^2 \ mod \ n$
10: **if** $b = n - 1$ **then**
11: go to step 18
12: **end if**
13: **if** $b = 1$ **then**
14: Output "Composite" and stop
15: **end if**
16: **end for**
17: Output "Composite" and stop
18: Next j
19: **end for**
20: *Output*"*Prime*"

Natural numbers that are greater than 1 and cannot be formed as a product of two smaller natural numbers are said to be prime. Natural numbers that are greater

k	t	k	t	k	t
160	34	202-208	23	335-360	12
161-163	33	209-215	22	361-392	11
164-166	32	216-222	21	393-430	10
167-169	31	223-231	20	431-479	9
170-173	30	232-241	19	480-542	8
174-177	29	242-252	18	543-626	7
178-181	28	253-264	17	627-746	6
182-185	27	265-278	16	747-926	5
186-190	26	279-294	15	927-1232	4
191-195	25	295-313	14	1233-1853	3
196-201	24	314-334	13	1854-up	2

Table 1.2: Number of trials (t) of Miller-Rabin test to generate prime numbers of bit size (k) with acceptable error probability [1]

than 1 and are not prime are known as composite numbers. The Miller-Rabin test as given in Algorithm 1 [1] will determine if n, is prime or composite with a small probability of error. To minimize the error probability, the Miller-Rabin test is run multiple times. Usually an error probability less than 2^{-100} is accepted. To achieve an error probability less than 2^{-100} for a 1024-bit positive integer n, the Miller Rabin test has to be run at least 4 times [1] as can be seen from Table 1.2.

1.6 Montgomery arithmetic

Modular multiplication and modular exponentiation using repeated modular multiplication are operations most commonly used in Public Key Cryptography (PKC) such as RSA and ECC. Implementing modular operations for Public Key Cryptography is a great challenge for both hardware and software platforms, as it involves computation with integers of large precision, for instance, 1024 bit integers for RSA. Montgomery arithmetic [17] is usually used to implement such modular operations in PKC. Montgomery arithmetic replaces expensive divisions with simple shift and add operations.

Let the modulus n be a k-bit integer. The Montgomery multiplication algorithm calculates,

$$MM(a,b) = a \cdot b \ \cdot r^{-1} \ mod \ n \tag{1.11}$$

where a and b are two integers such that $a, b < n$ and r such that $gcd(n,r) = 1$. Modular algorithm performs divisions by r, instead of n. Hence, r is usually chosen to be a power of 2, as on general purpose computers, divisions by power of 2 is a fast operation. We choose r to be 2^k. If n is odd, then any power of 2 as r will be relatively prime to n. r^{-1} is the inverse of r modulo n. Given two integers a, b and modulus n, to compute $t = a \cdot b \ mod \ n$ using Montgomery method, the inputs a and b are converted into Montgomery by calculating their respective n residues,

$$\bar{a} = a \cdot r \ mod \ n$$

$$\bar{b} = b \cdot r \ mod \ n$$

The Montgomery product of the two n-residues $\bar{a}$ and $\bar{b}$ will be,

$$\begin{aligned} \bar{c} &= MM(\bar{a}, \bar{b}) \\ \bar{c} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \ mod \ n \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \ mod \ n \\ &= c \cdot r \ mod \ n \end{aligned}$$

In order to retrieve c from its n residue form, we compute the montgomery product of $\bar{c}$ with 1.

$$\begin{aligned} MM(\bar{c}, 1) &= \bar{c} \cdot 1 \cdot r^{-1} \ mod \ n \\ &= c \cdot r \cdot r^{-1} \ mod \ n \\ &= c \ mod \ n \end{aligned}$$

The generic Montgomery multiplication algorithm [18] is given in Algorithm 2.

Algorithm 2 Montgomery multiplication algorithm [18]

Ensure: $MM(\bar{a}, \bar{b})$
1: $t \leftarrow \bar{a} \cdot \bar{b}$
2: $u \leftarrow (t + (t \cdot n' \ mod \ r) \cdot n)/r$
3: **if** $u \geq n$ **then**
4: **return** $u - n$
5: **else**
6: **return** u
7: **end if**

Modular multiplications and divisions with modulus r is very fast in comparison to computations with modulus n, as the former involves only shift operations if r is a power of 2. Conversion from integer to n residue, and vice versa are still time consuming, hence Montgomery method is not preferred for calculation of single modular multiplication. Montgomery arithmetic is advantageous if a large number of modular multiplications have to be performed with the same modulus consecutively, as is the case for modular exponentiation. The generic Montgomery exponentiation algorithm [18] for calculating $x = a^e \ mod \ n$ is given by Algorithm 3 below.

Algorithm 3 Montgomery Exponentiation algorithm [18]

Ensure: $ME\ (a, e, n)$
1: $\bar{a} \leftarrow a \cdot r \ mod \ n$
2: $\bar{x} \leftarrow 1 \cdot r \ mod \ n$
3: **for** $i \leftarrow j - 1$ *to* 0 **do**
4: $\bar{x} \leftarrow MM(\bar{x}, \bar{x})$
5: **if** $e_i = 1$ **then**
6: $\bar{x} \leftarrow MM(\bar{x}, \bar{a})$
7: **end if**
8: **end for**
9: $x \leftarrow MM(\bar{x}, 1)$
10: **return** x

It is important to note that there are plenty of software and hardware implementations for Montgomery multiplication and exponentiation in the literature.

Since the number of bits in a,b and n are usually too high to be processed as such in processors, they are usually broken up into words and computations are done on a word by word basis. However, from a system level perspective it is more feasible to consider word based algorithms whose word size match the processor word size.

CHAPTER 2

PERFORMANCE PROFILING OF RSA SYSTEM ON EMBEDDED DEVICES

2.1 Determining hotspot function

Let us define *HF Rate* or *Hotspot Function (bottleneck function) Rate* of a function to be the percentage of overall execution time that is dedicated to the function in consideration. Amdahl's law, given by,

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

states that $Speedup_{overall}$ can be observed in the overall algorithm, if a function, which takes up $Fraction_{enhanced}$ portion of the overall algorithm, is improved by $Speedup_{enhanced}$. Note that *HF Rate* is equivalent to $Fraction_{enhanced}$. If the *HF Rate* of a function is too low, then the overall speedup cannot be significant even if the performance of the function can be improved. The higher the *HF Rate*, the higher would be the impact on overall speedup if the performance of the function can be improved. However, the candidate for hotspot function should not be the entire algorithm itself, as it would correspond to higher hardware area cost. Thus, a function can be considered as a good candidate for hardware acceleration and can be chosen as the *hotspot* function if it consumes substantial amount of the overall execution time and has minimal hardware area [19].

2.2 Methodology

In order to figure out a good candidate for acceleration, it is essential to find out the runtime of different sections of the code. The runtime of a particular section of the code is impacted not only by the time taken to run the particular code section once, but also by the number of times the code section is ran.

In order to analyze the behaviour of RSA encryption, decryption and key generation algorithms, they are run in different platforms containing processors of varying capabilities. One platform chosen is a Linux machine running an Intel Dual-Core i5-4210U CPU @ 1.7GHz with 4GB memory. Another platform is a Raspberry Pi 3 Model B+ machine with a Broadcom BCM2837B0 Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz, 1GB LPDDR2 SDRAM. Another platform chosen is Nios II 32-bit soft-core processor configured on an Altera FPGA in De2-i150 board, with on chip memory of 400kB.

RSA Algorithms that are run in these platforms are taken from the mbedtls cryptographic library which is an open source library written in C, available under Apache 2.0 License. The library contains algorithms which are lightweight, loosely coupled and portable to multiple platforms [20].

2.3 Experiment 1 : RSA runtime analysis across platforms

RSA key generation, encryption and decryption are run repeatedly in the three platforms mentioned to determine the average time taken to complete the operations. In Intel Linux machine and ARM Raspberry Pi, the operations were run 1000 times whereas in the Nios II processor the operations were run 50 times. The results obtained are tabulated in Table 2.1.

A number of observations could be made from the results. In Intel Linux machine, the time taken to complete operations are in milliseconds, whereas in ARM Raspberry Pi it takes time an order of magnitude more than the Intel Linux machine. Similarly

Operation	Intel*(ms)*	Rasp Pi*(ms)*	NiosII*(ms)*
Encryption	0.47	3.73	95
Decryption	5.73	93.76	2621
Key generation	251.99	9453.68	102100

Table 2.1: RSA runtime analysis across platforms

in Nios II processor, it takes time atleast an order of magnitude more than the ARM Raspberry Pi. As the processing power of the platform decreases, the performance of the RSA operations also decreases, as it can be observed in fig 2.1.

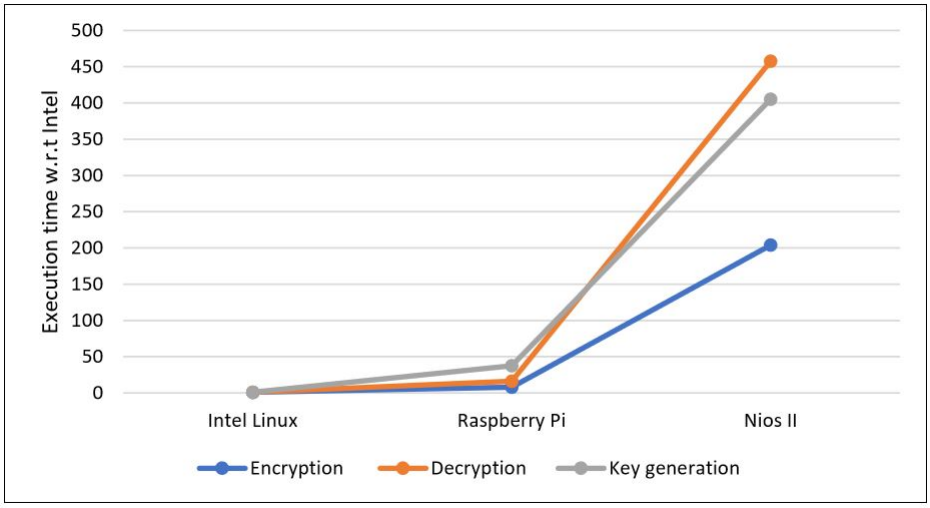

Figure 2.1: Performance across platforms

Time distribution of RSA operations across platforms is given in pie charts in figures 2.2, figure 2.3 and figure 2.4. If time taken by all RSA operations is assumed to be x, it can be seen that on average, 98 percentage of x is consumed by key generation, 1.9 percentage of x is consumed by decryption and 0.1 percentage of x is consumed by encryption.

The RSA algorithm is commonly used for session key distribution and digital signature verification, where the plaintext or message (which is usually a key to be used for data encryption) length is short. In addition to this, our application, in

particular, demands a new set of keys for each session. Hence, the overall RSA process is key generation to generate a new pair of keys, encryption or decryption involving message of short length. For other applications, key generation may not take place per session and hence key generation could amortize encryption/decryption.

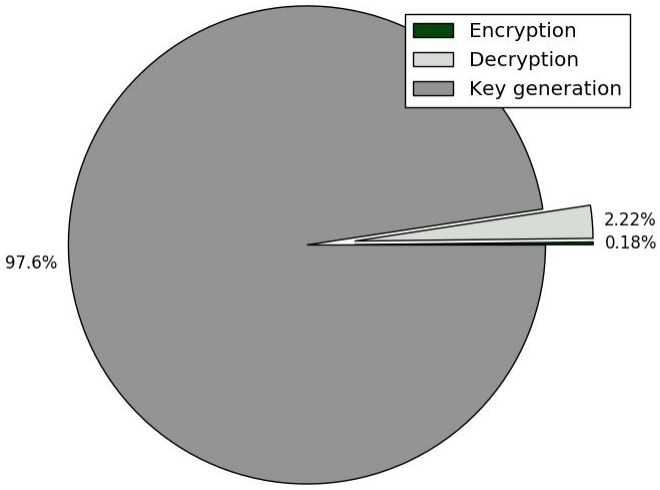

Figure 2.2: Time distribution for RSA operations - Intel Linux

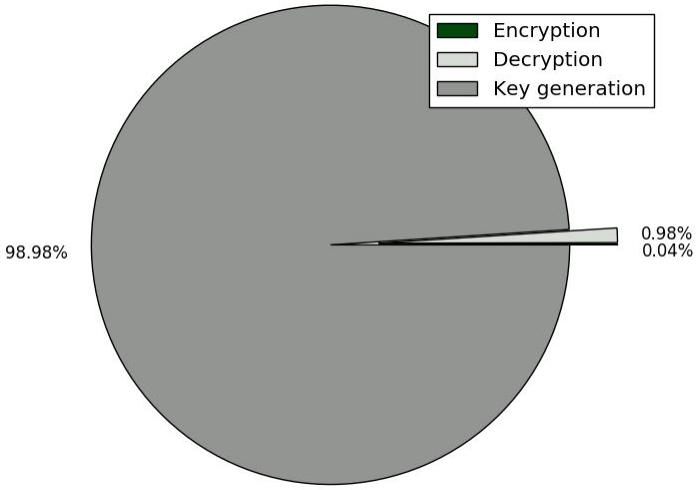

Figure 2.3: Time distribution for RSA operations - ARM Raspberry Pi

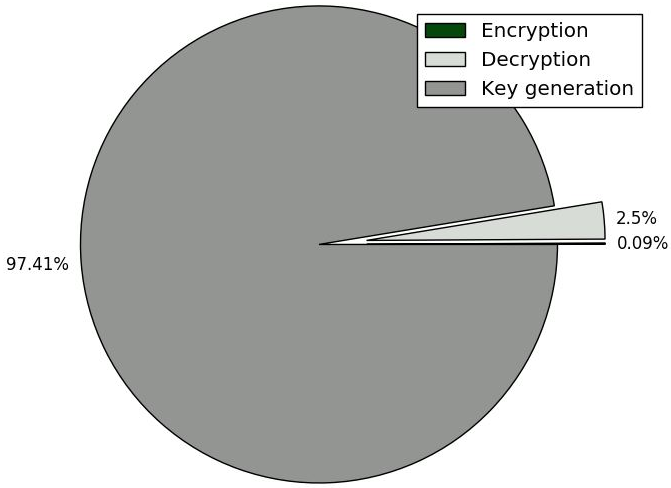

Figure 2.4: Time distribution for RSA operations - Nios II

2.4 Experiment 2 : Analyzing RSA key generation runtime

From 2.1, it can be seen that key generation takes up 98% of time and this algorithm has to be analyzed for selecting a good hotspot candidate.

The key generation algorithm is first analyzed to determine potential candidates. Key generation algorithm is given in section 1.3.1. In practice, however, the algorithm is carried out in a different manner as mentioned in section 1.4. The public verification exponent e is selected prior to generating the primes, p and q and the private signature exponent d. As in common practice, the e value is chosen to be 65537.

The number of calls to the primality test algorithm when the key generation algorithm is run 1000 times is tabulated in Table 2.2. It is observed that in order to generate suitable prime numbers p and q, primality test has to be run a large number of times and it occupies a significant portion of the key generation algorithm.

RSA key generation algorithm is run 1000 times and the time taken for suitable prime generation is measured and reported in Table 2.3. The RSA key generation is run 1000 times to enable profiling. In those 1000 runs, on an average the number of calls to RSA ke y generation is 1, suitable prime generation is 2, and the primality test algorithm is 115. Calculation of modulus from suitable primes, computation of

Function	*Intel Linux*		*Raspberry Pi*	
	Number of calls	Average	Number of calls	Average
RSA key generation	1000	1	1000	1
Suitable prime generation	2000	2	2000	2
Primality tests	115150	115.2	114672	114.7

Table 2.2: Number of calls to primality test

private key exponent take negligible amount of time, less than one percentage, as compared to generation of suitable primes.

Platform	Suitable Prime gen.	Key gen.	Percentage
Intel	0.251 ms	0.252 ms	99.48
Raspberry Pi	9.42 ms	9.45 ms	99.68
Nios	101.65 s	102.1 s	99.56

Table 2.3: Key generation analysis

Across platforms, almost 99.573% of the key generation time is taken for suitable prime generation. This percentage dependency goes higher as the time taken for key generation increases as it can be seen from from figures 2.5 and 2.6. Thus, it can be concluded that speeding up suitable prime number generation will help speed up the key generation process.

2.5 Experiment 3 : Analyzing suitable prime generation runtime

The Miller-Rabin prime number generation algorithm is used to check if numbers generated randomly are prime according to Algorithm 1. Once numbers are determined to be prime, they are checked to determine if they satisfy the inequalties mentioned in equations 1.6, 1,7 and 1.8. Until they do so, the Miller-Rabin prime generation algorithm is run over and over again to generate prime numbers p and

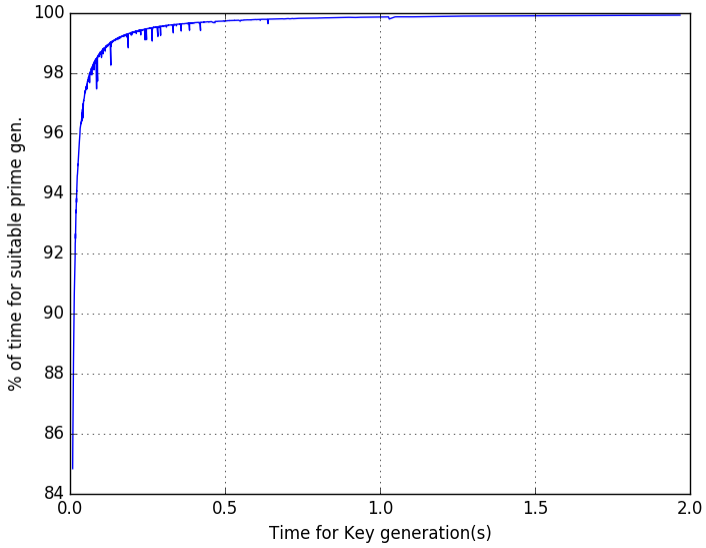

Figure 2.5: Time taken for suitable prime generation vs time taken for key generation in Intel Linux machine

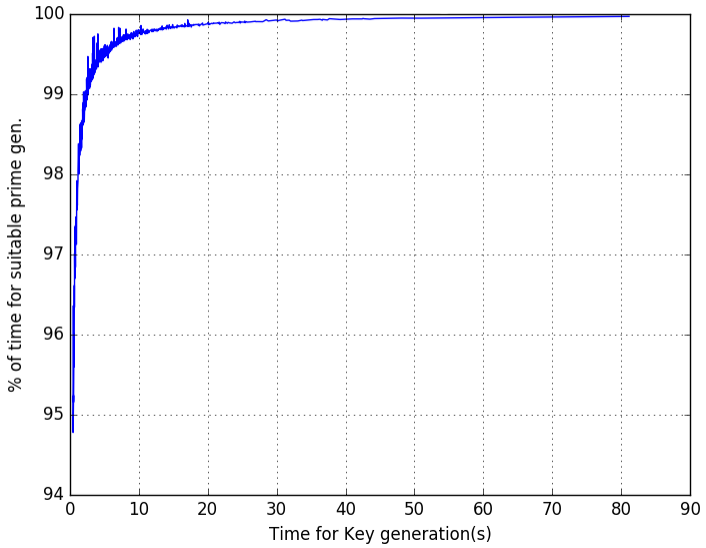

Figure 2.6: Time taken for suitable prime generation vs time taken for key generation in Raspberry Pi

q that satisfy this criteria. The suitable prime generation algorithm which involves cycles of generation of primes using Miller-Rabin algorithm and checking of primes for suitability with selected public key exponent is profiled using a profiling tool to aid in the determination of hotspot function. If multiple rounds of Miller Rabin are to be avoided, p and q could be chosen first, and then the modulus, and then e or d determined. This is adopted in the original RSA key genration algorithm, as detailed in section 1.3.1. However, in practice, e is chosen to be 65537, a fixed value [14]. With e as 65537, which is 10000000000000001 in binary, it makes the RSA encryption or signature verification process easier for computation. Having a fixed public key exponent value of 65537 does not have security implications. If RSA key generation algorithm in practice is adopted, which is detailed in section 1.4, multiple rounds of Miller Rabin cannot be avoided.

Since the mbedtls library is loosely coupled, the algorithm is composed of smaller functions, making it easier for analysis purposes. We use a popular profiling tool GNU Profiler or *gprof* to analyze a program and determine where the key generation program spends its time. To generate profile information for our program, the program is compiled and linked with profiling enabled using *-pg* option while running the compiler.

gprof provides information about a program in the form of *Flat profile* and *Call graph*. Flat profile gives information about the time spent in executing each function by displaying percentage of total execution time the function takes *(% time)*, time taken by the function in seconds *(self seconds)*, and total number calls to the function *(calls)* among other fields for each function. The call graph provides information on which function *(parent function)* calls which other function *(child function)*, the number of times it is called *(called)*, the time taken by the function *(self)* and its children*(children)*. It helps in determining if the time is spent in the function itself, or if the time is spent in executing its subroutines or children functions.

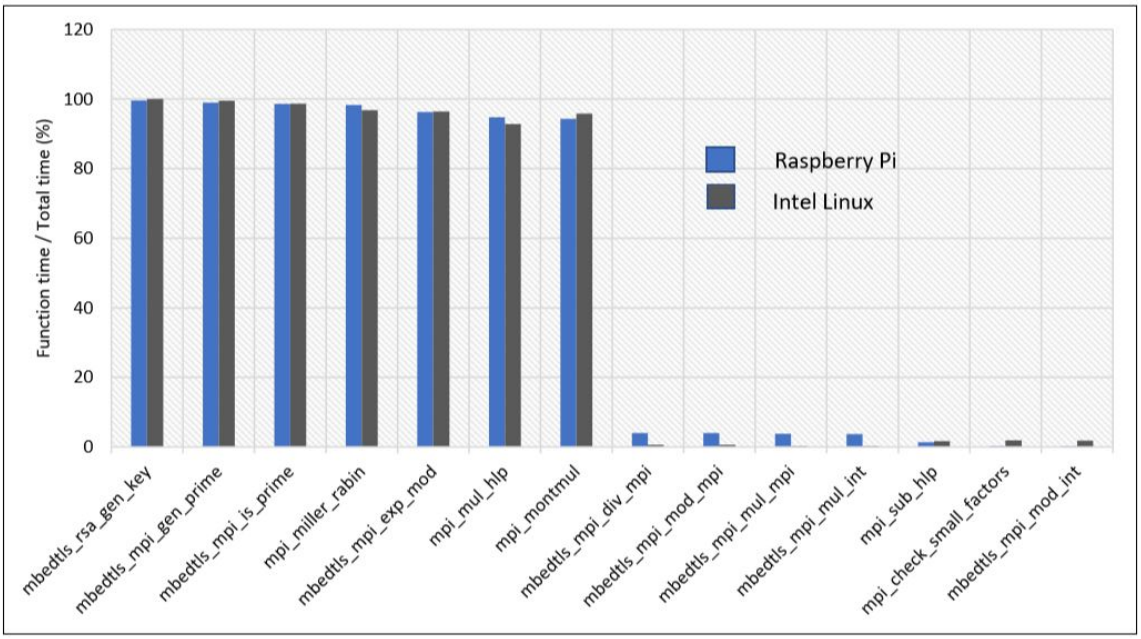

Figure 2.7: Function execution time

Information from the profiling tool, reveals the top functions that consume the most amount of time to be :

1. *mbedtls_rsa_key_gen* : This function is called to compute a pair of 2048-bit RSA keys for a public key exponent value 65537.

2. *mbedtls_mpi_gen_prime* : This function generates suitable prime numbers for a public key exponent value of 65537. It is a child function of *mbedtls_rsa_gen_key*.

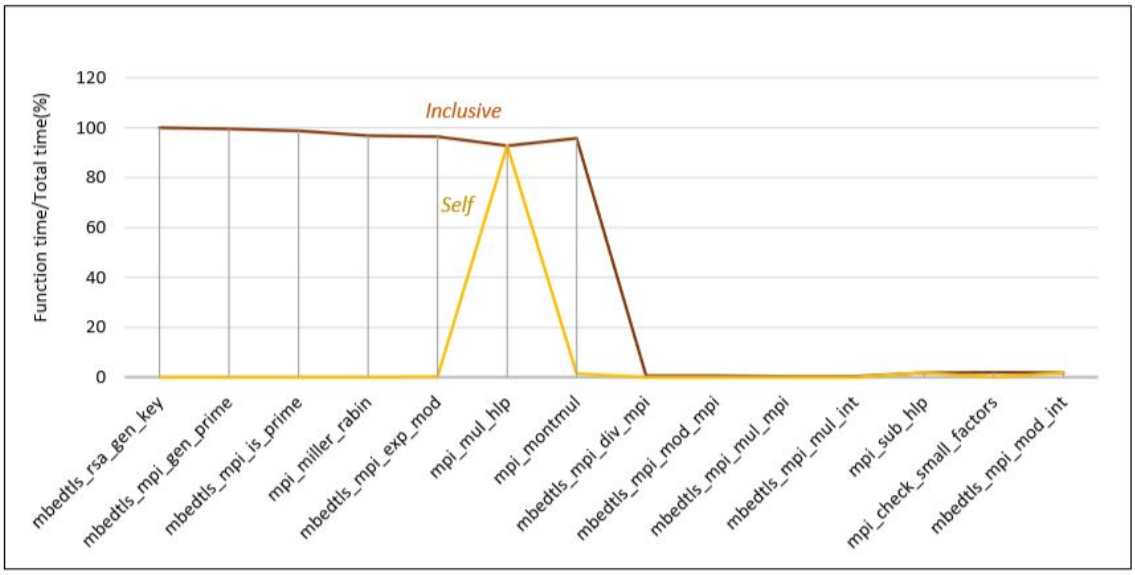

Figure 2.8: Function execution time Intel

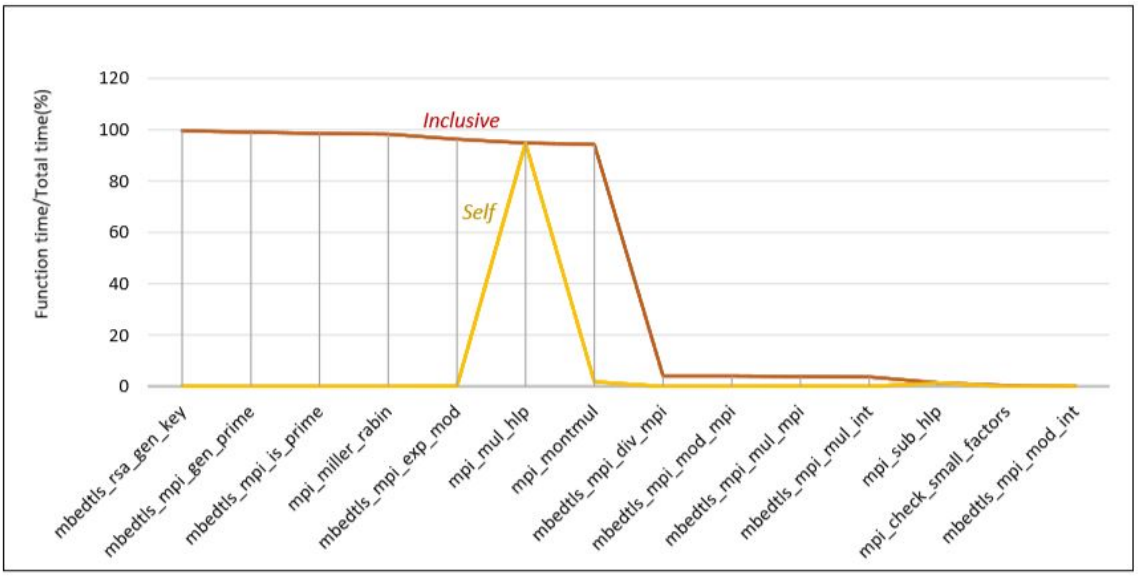

Figure 2.9: Function execution time Raspberry Pi

3. *mbedtls_mpi_is_prime* : This function checks if the given number is prime or not, first by checking for small factors and then by running Miller-Rabin primality test. It is a child function of *mbedtls_mpi_gen_prime*.

4. *mpi_miller_rabin* : This function implements the Miller Rabin primality test for a given integer and determine if the given integer is prime or not. Since generating 2048-bit keys require 1024-bit prime numbers, the input for Miller-Rabin primality test will be a 1024-bit integer.

5. *mbedtls_mpi_exp_mod* : This function performs 1024-bit modular exponentiation and is a child function of *mpi_miller_rabin*.

6. *mpi_mul_hlp* : This function performs the operation given below.

---

for $j \leftarrow\ 0\ to\ s-1$ **do**
$(C, t[j]) \leftarrow t[j] + x[j] * Y + C$
end for

The function aids in multi-precision multiplication and multi-precision Montgomery modular multiplication and hence is called by two functions, namely, *mbedtls_mpi_mul_mpi* and *mpi_montmul*. *mpi_montmul* is responsible for 99.74% of calls in Intel Linux machine and 96.15% of calls in Raspberry Pi.

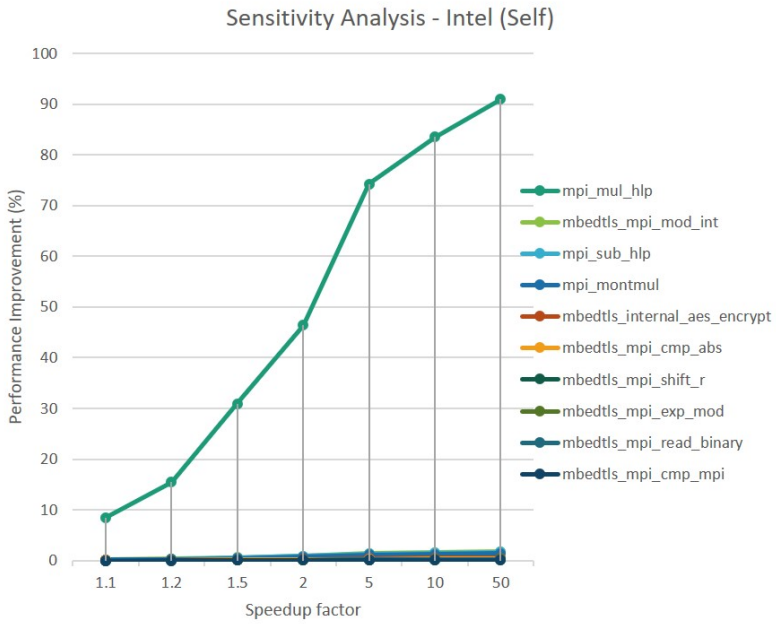

Figure 2.10: Sensitivity Analysis for various functions - Intel Linux

7. *mpi_montmul* : This function performs Montgomery modular multiplication for 1024-bit integers. This is a child function of *mpi_montred*, which is used to implement Montgomery reduction and a child function of *mbedtls_mpi_exp_mod*.

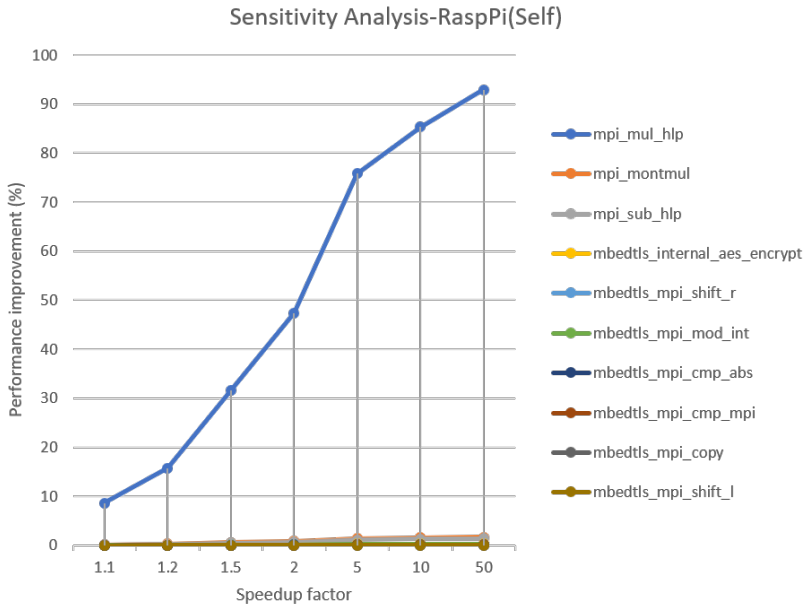

Figure 2.11: Sensitivity Analysis for various functions - Raspberry Pi

Time taken by a function can be due to the function itself (self execution time) and/or due to the children functions. A good candidate for hotspot function will be a function which has a high self execution time. The most time consuming functions listed above were profiled to determine the function that has the highest self execution time. *mpi_mul_hlp* was found to have the highest self execution time as evident from figures 2.8 and 2.9. The values have been tabulated in 2.4. Other functions listed were observed to be parent functions of the *mpi_mul_hlp*.

Sensitivity analysis is carried out to determine the performance improvement that would be observed on the overall algorithm for various speedup factors for candidate hotspot functions. Each of the function is expected to improve by 10%, 20% , 50%, 100% etc, and the impact on the overall algorithm is computed in the sensitivity

Function Name	**Intel Linux**		**Raspberry Pi**	
	Incl.time(%)	*Self time(%)*	*Incl.time(%)*	*Self time(%)*
mbedtls_rsa_gen_key	100	0	99.6	2.66×10^{-5}
mbedtls_mpi_gen_prime	99.5	10.5×10^{-4}	99	7.45×10^{-4}
mbedtls_mpi_is_prime	98.7	10.5×10^{-4}	98.6	2.4×10^{-4}
mpi_miller_rabin	96.8	0	98.3	7.45×10^{-4}
mbedtls_mpi_exp_mod	96.4	0.23	96.3	0.052
mpi_mul_hlp	**92.8**	**92.8**	**94.8**	**94.8**
mpi_montmul	95.8	1.41	94.3	1.74
mbedtls_mpi_div_mpi	0.6	0.015	4	0.013
mbedtls_mpi_mod_mpi	0.6	0	4	1.32×10^{-4}
mbedtls_mpi_mul_mpi	0.3	0.029	3.8	0.042
mbedtls_mpi_mul_int	0.3	0.008	3.7	0.002
mpi_sub_hlp	1.7	1.7	1.4	1.4
mpi_check_small_factors	1.9	0.013	0.3	0.005
mbedtls_mpi_mod_int	1.8	1.8	0.2	0.2

Table 2.4: Function execution time - inclusive and self

analysis process. For both the platforms, for all values of speedup, the *mpi_mul_hlp* function standouts to have the most impact on the overall algorithm as it can be seen from figures 2.10 and 2.11. The hotspot function is determined to be the operation executed by *mpi_mul_hlp*.

Thus, the RSA algorithm is profiled and bottleneck identification is complete. The function implemented by *mpi_mul_hlp* is determined to be the bottleneck for RSA algorithm and is chosen to be most suited for hardware acceleration.

CHAPTER 3

ACCELERATOR DESIGN

In this chapter, the system level considerations for accelerator design and their relevance are discussed. The design approach adopted in this work, the accelerator architecture proposed and the architectures used for comparison are also discussed. Some previous works on Montgomery multipliers for reconfigurable hardware are also summarized in this chapter.

3.1 System level considerations for accelerator design

Accelerators are nothing but the hardware components designed to execute a specific functionality in an SoC. They are capable of improving performance (by 10-100x) and lowering energy consumption (by 100-1000x) as compared to software implementations in processors [21]. Implementation in hardware leads to faster systems, however, implementation in software is cheaper and keeps the design flexible. Design of an SoC system with accelerator begins with functional specification of the entire system allowing for a functional verification. It then proceeds to partitioning design into hardware and software while accounting for performance, cost and flexibility.

The microarchitecture of a typical accelerator is given in figure 3.1[2]. It consists of a data-path and a controller. Controller is a finite state machine which controls the execution of the desired accelerator function based on a set of conditions. It controls execution of operations in the hardware resources, also called the datapath, in a given clock cycle. The datapath operate on data obtained from memory either local or external. The accelerator proposed in this work has a similar microarchitecture

where the data for the accelerator is stored in a memory external to the accelerator. Since the accelerator implements a sequential design, the control of the datapath to generate a Montgomery product is split between the accelerator and the processor. Accelerators can be located either inside the processing core (tightly-coupled) or outside (loosely-coupled).

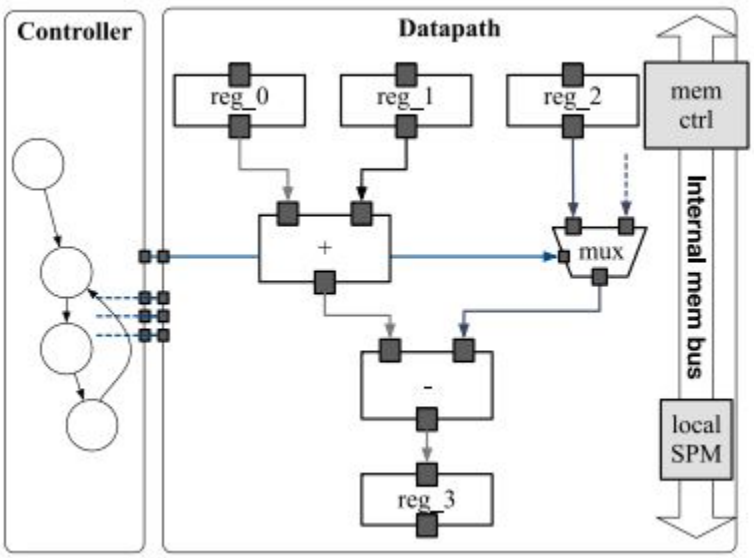

Figure 3.1: Microarchitecture of a typical accelerator [2]

State of the art accelerators use scratchpad memory and Direct Memory Access to provide fixed-latency data transfer and memory access. Typically it is observed that On-chip memory consumes about 40-90% of accelerator area. DMA engines are responsible for moving data between general purpose cores and accelerators [22]. Loosely-coupled accelerators are suitable for computation on large-data sets while tightly coupled accelerators are for fine-grain computations on small data sets.

To account for increasing complexity of SoCs and the accelerators, design effort should shift from RTL or Register Transistor Level Design to System Level Design. The system constraints play a vital role in deciding design of the accelerator and eventually overall performance.

Consider a scenario in which an accelerator datapath is capable of processing data at a rate that is faster than the rate in which data can be transferred over the system interconnect. In such a case, the logic area in the accelerator is not put to optimal use, which could be rectified by a change in the hardware resources by increasing memory or by changing the architecture of the accelerator in order to be more suitable for system specifications. Variations in software code leading to slightly different versions of the hotspot function should also be considered before narrowing down an architecture to be suitable for hardware acceleration for the system environment. For instance, let us consider that the function to be accelerated is given be $f(x, y, z, a, b) = x + y + z + a + b$. Assume two possible architectures for hardware accelerators,

$$a1(m, n) = m + n$$

$$a2(m, n, o) = m + n + o$$

Depending on the accelerator architecture chosen, $f(x, y, z, a, b)$ could be calculated by running the accelerator twice or four times. This demonstrates how change in software code on the processor is required depending on the accelerator architecture. The system might be capable of transferring 2 inputs or 3 inputs per clock cycle to the accelerator, depending on which the accelerator architecture could be chosen. This demonstrates how change in the system constraints can affect the choice of accelerator architecture.

More often than not, system constraints are disregarded while designing hardware accelerators, leading to non-optimal designs or under-performance. In order to obtain an optimal design, the design space should be analyzed to optimize the architecture of an accelerator and evaluate different trade-offs in terms of performance and costs.

3.2 Hotspot function and relation to Montgomery multiplication

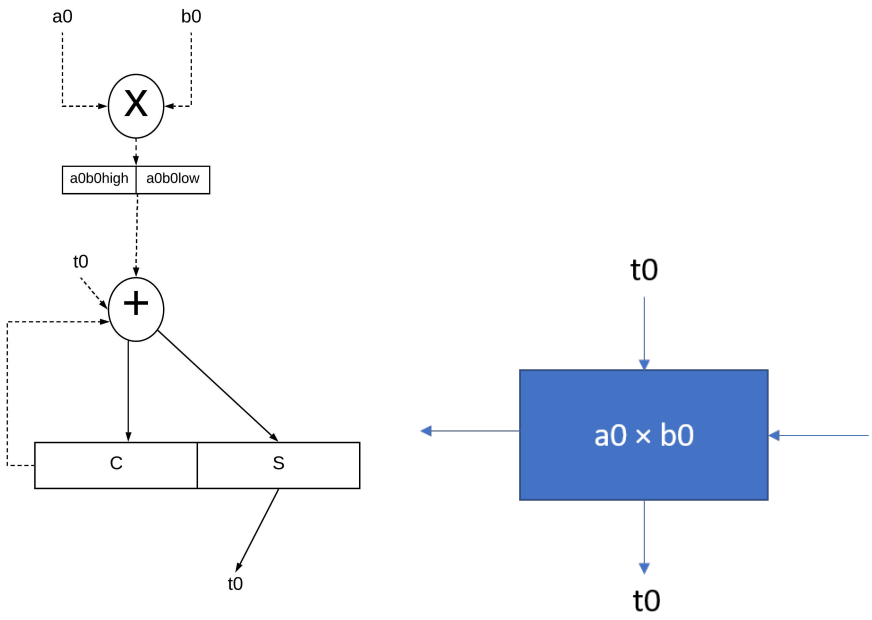

Figure 3.2: Minimal architecture to implement mpi_mul_hlp (left) and block representation (right)

The mpi_mul_hlp function is given in the following algorithm.

---

for $j \leftarrow\ 0\ to\ s-1$ **do**

$(C, t[j]) \leftarrow t[j] + x[j] * Y + C$

end for

From the experiments in the previous chapters, we observe that *mpi_montmul* function is responsible for almost 97.98% of calls to *mpi_mul_hlp* function. It forms the inner loop of the word based Montgomery multiplication algorithms. If *mpi_mont_mul* can be represented diagramatically by the block diagram shown in 3.3, then *mpi_mul_hlp* represents a row of operation.

mpi_mul_hlp function directly corresponds to inner j loops of CIOS (Coarsely Integrated Operand Scanning) algorithm [18]. The CIOS Montgomery multiplication

is a word based modular multiplication technique where the large integers are broken into smaller words of bit length usually same as the processor word size. The inner loop 1 corresponding to lines 3 to 6 of Algorithm 5 and inner loop 2 corresponding to lines 12 to 15 of Algorithm 5 implement the same algorithm but on different inputs. The generic inner loop is given in Algorithm 6.

If to be implemented sequentially, the minimal hardware architecture that can perform this operation would be a multiplier unit followed by and adder unit capable of adding 2 single precision number to the result of multiplication which is of double precision. The architecture required to implement the same can be represented by a block as shown in 3.2.

3.3 Previous works

In order to implement RSA systems on hardware, many previous works have proposed Montgomery implementations on FPGAs [23], [24], [25], [26], [27] . The Montgomery multiplication algorithm, introduced by Peter Montgomery in 1985, is based on a series of addition and shift operation and it replaces the cumbersome trial division by modulus method to compute modular multiplication. There are many variants of Montgomery multiplication algorithms - radix 2, systolic, higher radix implementations. A lot of previous works have focused on Montgomery multipliers as stand alone circuits.

When working with long operands for RSA (typically of size 1024 or 2048 bits), the performance is affected by long carry propagation. Previous works on Modular exponentiation using Montgomery multiplication have made use of carry save format to avoid long carry propagation delays. In these works, the inputs and outputs are represented in binary form, but the intermediate results are in in carry-save format which necessitates format conversion at the end of each multiplication. In other cases like [28], the inputs and outputs are maintained in carry-save format, and conversion

to binary is performed only to obtain the final result of modular exponentiation at the expense of additional registers to store intermediate operands. In [28], an implementation to reduce hardware resources have been proposed with focus on final output conversion circuit to convert form CSA to binary form. Instead of 1024 bit CPA (Carry Propagation adders), a 32-bit CPA is used, with shift registers to feed input values to the same. The Montgomery multiplication however still works on the entire $n - bit$ operands with 2 double CSAs generating result in CSA format in n clock cycles. In [29], a complexity effective multiplier version has been proposed, which stores intermediate results from one adder to a look up table which feeds a CSA adder structure. In all of these designs, the adders to facilitate multiplication are the same size as operand length n. In [30], 128 and 256 bit CIOS (Coarsely Integrated Operand Scanning) architecture was implemented in Xilinx Virtex2 family of FPGAs and compared with FIOS (Finely Integrated Operand Scanning), SOS (Separated Operand Scanning) architecture implementations for area and throughput. In [31], a parametric design, for modular multiplication, exponentiation and prime tester is mentioned. The algorithm, which is tunable and scalable to meet the area/speed requirements operates on individual bits of the operands. In [32], energy efficient architecture using BRFA(Barrel Register Full adder) and clock gating technique is suggested. They propose a modified MM42 architecture using double carry save adder structure for 1024 bit radix-2 multiplication with additional circuitry for lookup logic(LU), superfluous CSA and register write operation bypassing. Most of these works have focused on Register Transfer Level design, disregarding system level considerations.

3.4 Word based Montgomery multiplication algorithm for accelerator

A generic Montgomery Multiplication algorithm is given in Algorithm 4. Line 1 of Algorithm 4 represents integer multiplication. The rest accomplishes Montgomery reduction. In the generic algorithm, the operands are of length 1024 bits.

Algorithm 4 Montgomery multiplication algorithm [18]

Ensure: $MM(\bar{a}, \bar{b})$
1: $t \leftarrow \bar{a} \cdot \bar{b}$
2: $u \leftarrow (t + (t \cdot n' \ mod \ r) \cdot n)/r$
3: **if** $u \geq n$ **then**
4: **return** $u - n$
5: **else**
6: **return** u
7: **end if**

From a system level perspective, we propose that accelerators that are tightly coupled with the processor work best with word based multiplication methods. The focus of this work is on accelerating word based Montgomery multiplication methods. Since the operand size is of multiple precision, they are split into words of single precision. In a processor, any operation on this multi-precision operands would involve computations on single precision words which are then combined together to obtain the final result. Since the processor is capable of operating only in terms of words, it is desirable that an accelerator that works closely with the processor also implement word based algorithms.

Suppose the processor word size is 8 bits. A 1024 bit operand is now split into 128 words. If the accelerator is such that it is capable of operating on all of 1024 bits, then the accelerator is stalled for 128 * x cycles, where x is the delay in transferring single operand word to the accelerator. However, with word based algorithms, computation can begin as soon as the lower precision words (word based algorithms usually start computation with lower precision words) of the operands are made available. Word

based algorithms also generate results in word-units incrementally, thus not stalling the processor to wait for result until the complete or final result (multi-precision number) is computed.

3.4.1 Representing Montgomery word based algorithm

The modulus n is a k-bit integer where $2^{k-1} \leq n \leq 2^k$ and r is 2^k. In order to compute Montgomery reduction algorithm, n' is required which is obtained by $r \cdot r^{-1} - n \cdot n' = 1$. In all word based algorithms the operands a, b and the modulus n are r bit integers. They are split into s words, where w is the word size. The radix W is 2^w, where $r = 2^{sw}$. In all word based algorithms, a word of a multiprecision integer is represented by its index. For instance $a[0]$ represents lowest significant word of a.

The data flow and data path of a four word Montgomery multiplication is shown in 3.3. White boxes represent operations for integer multiplication. Shaded boxes represent operations for Montgomery reduction. Assume a and b are multi-precision integers, where a consists of four words indexed by j: $a[0], a[1], a[2], a[3]$and b consists of four words indexed by i : $b[0], b[1], b[2], b[3]$. White box represents multiplication of words of a and b represented by their indices, and addition of inputs t_{in}, c_{in} generating outputs t_{out} and c_{out}. Shaded box represents montgomery reduction m_{in} and $n'[j]$ are multiplied generating t_{in}, c_{in} generating outputs t_{out} and c_{out}. m_{in} which can be calculated from $t[0]$ and $n'[0]$. Adding the partial results of all the boxes, the Montgomery product can be obtained by keeping the upper half of the final result.

There are different sequential implementation schemes for Montgomery multiplication which gives rise to various word based algorithms. The difference in these methods comes from the sequence to calculate boxes. If the white boxes are calculated first, and then the all the shaded boxes, row-wise, then it is called Seperated Operand Scanning (SOS) method. If a row of white boxes are calculated and then a row of shaded boxes are calculated then it is called Coarsely Integrated Operand

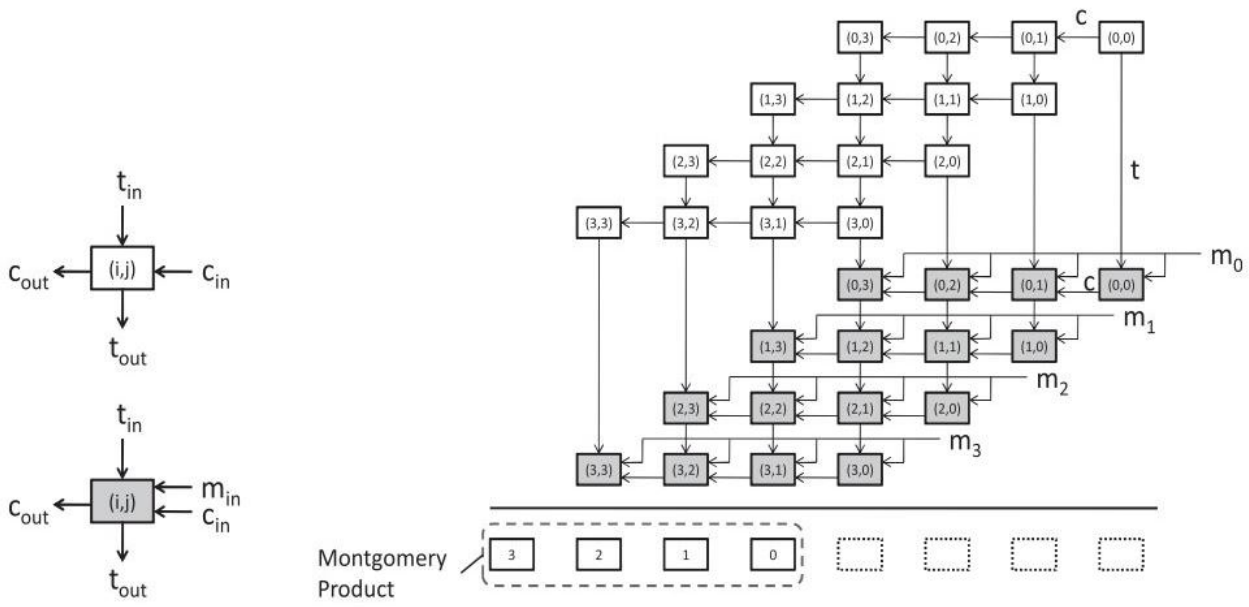

Figure 3.3: Block diagram representing dataflow of Montgomery multiplication. White box represents integer multiplication block. Shaded box represents montgomery reduction block [3]

scanning (CIOS) method. If a white box is calculated and then a shaded box is calculated, it is called Finely Integrated Operand Scanning (FIOS) Method.

3.5 Accelerator design approach

Minimal datapath circuit to implement the *mpi_mul_hlp* includes multiplication of two single precision words followed by addition of the result to 2 single precision words. This operation is generally termed as 'Multiply accumulate' and is prevalent in cryptographic algorithms. Some works have considered instruction set extensions to include Multiply Accumulate (MULACC) instruction on MSP430, ATMega, CortexM0+ to facilitate this operation [33], [34], [35].

In designing the accelerator, one aims to look for hardware that is slightly more sophisticated and higher in functionality than the minimal architecture for *mpi_mul_hlp* and is capable of accelerating the montgomery multiplication method and in effect

the overall RSA key generation operation. The accelerator design proposed in this work satisfies this criteria. We introduce an algorithm to implement Montgomery multiplication and the minimal architecture required to implement the same. The accelerator hardware in consideration can be equated to the loop body of the word-based algorithms. The loop body is iterated a number of times with different inputs to implement the algorithm. Similarly, the accelerator has to be run multiple times to complete one round of Montgomery multiplication. The processor is responsible to provide appropriate inputs to the accelerator to facilitate this. The datapath to compute Montgomery multiplication belongs to the accelerator, while the control circuitry is split between the processor and the accelerator.

Throughout this study, we focus on operand scanning methods, since the hardware for operand scanning methods are the most suited for hardware implementation, due to its uniform structure. We analyze our proposed design with similar and comparable architectures capable of implementing other word based algorithms like CIOS (Coarsely Integrated Operand Scanning) method and FIOS (Finely Integrated Operand Scanning Method).

Keeping the system level constraints in mind, we compare the designs in terms of area, latency in completing a Montgomery multiplication operation under input availability constraints. We assume the accelerators to have 2, 3 and 4 ports to transfer operands in and out of the accelerator and compare their performance in each case.

In the following sections, we analyze the CIOS, FIOS and proposed algorithm and describe the architecture of accelerators that implement CIOS, FIOS and the proposed algorithm.

3.6 Coarsely Integrated Operand Scanning (CIOS) algorithm

The CIOS algorithm is given in Algorithm 5. The algorithm obtains its name as it alternates between iterations of computing a row of integer multiplication followed by a row of montgomery reduction.

Algorithm 5 CIOS Algorithm [18]

1: **for** $i \leftarrow 0$ to $s-1$ **do**
2: $C \leftarrow 0$
3: **for** $j \leftarrow 0$ to $s-1$ **do** ▷ Inner Loop 1
4: $(C, S) \leftarrow t[j] + a[j] * b[i] + C$
5: $t[j] \leftarrow S$
6: **end for**
7: $(C, S) \leftarrow t[s] + C$
8: $t[s] \leftarrow S$
9: $t[s+1] \leftarrow C$
10: $C \leftarrow 0$
11: $m \leftarrow t[0] * n'[0] \bmod W$
12: **for** $j \leftarrow 0$ to $s-1$ **do** ▷ Inner Loop 2
13: $(C, S) \leftarrow t[j] + m * n[j] + C$
14: $t[j] \leftarrow S$
15: **end for**
16: $(C, S) \leftarrow t[s] + C$
17: $t[s] \leftarrow S$
18: $t[s+1] = t[s+1] + C$
19: **for** $j \leftarrow 0$ to s **do**
20: $t[j] \leftarrow t[j+1]$
21: **end for**
22: **end for**

Algorithm 6 Inner Loop of CIOS Algorithm

1: **for** $j \leftarrow 0$ to $s-1$ **do**
2: $(C, S) \leftarrow t[j] + x[j] * y + C$
3: $t[j] \leftarrow S$
4: **end for**

The minimal accelerator architecture to facilitate CIOS algorithm is given in figure 3.6. It consists of 2 multiplier units and associated adder units to compute partial products of first two indices in parallel. Using block diagram representation of dataflow as shown in figure 3.4, one can see that using the architecture, computation

Precision	**Word size = 8 bit**	**Word size = 16 bit**	**Word size = 32 bit**	**Word size = w bit**
Number of words	1024 / 8 = 128	1024/ 16 = 64	1024 / 32 = 32	1024 / w = s
Number of word-wise multiplications for Integer multiplication	64 (2 8 bit * 8bit) blocks * 128 iterations	32 (2 16 bit * 16 bit) blocks * 64 iterations	16 (2 32 bit * 32 bit) blocks * 32 iterations	s/2 (2 w bit * w bit) blocks * s iterations
Number of word-wise multiplications for Montgomery reduction	64 (2 8 bit * 8bit) blocks * 128 iterations	32 (2 16 bit * 16 bit) blocks * 64 iterations	16 (2 32 bit * 32 bit) blocks * 32 iterations	s/2 (2 w bit * w bit) blocks * s iterations

Table 3.1: Number of word-wise multiplications for Montgomery multiplication using CIOS accelerator

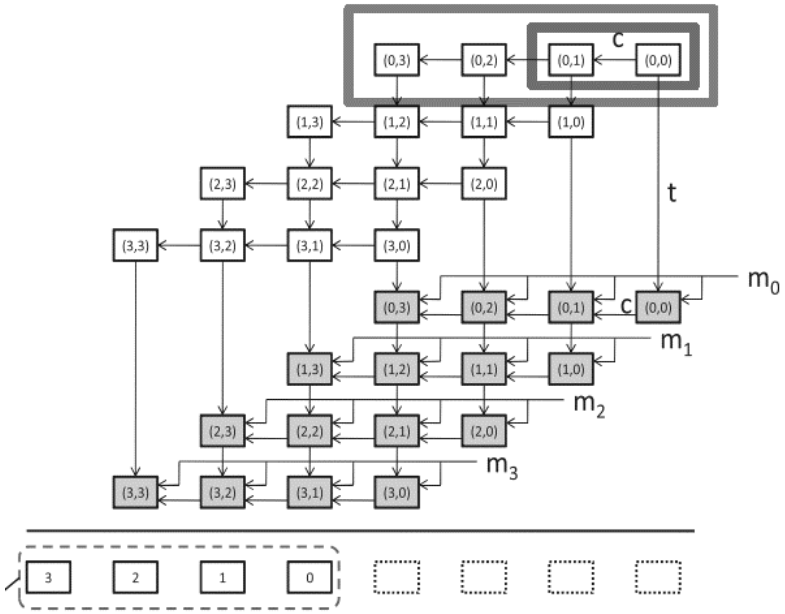

Figure 3.4: Block diagram representing dataflow of CIOS Montgomery multiplication

can proceed in 2 block units at a time. If integer multiplication (or montgomery reduction) is being computed, the 2 blocks compute integer multiplication (or montgomery reduction) results for consecutive indexes of a row in a concurrent manner.

The inputs required for CIOS loop accelerator are 2 words of a, 1 word of b and 2 words of t in Algorithm 6 and from figure 3.5. It produces 2 words of t as outputs. For the subsequent iteration/operation in the same row, the word b remains constant, but it requires 2 new words of a, and 2 new words of t.

The operations involved for one iteration are read of 2 words of a, 2 words of t, 2 word-multiplications, 2 3-word additions, storing results in 4 registers *C1*, *S1*, *C0*,

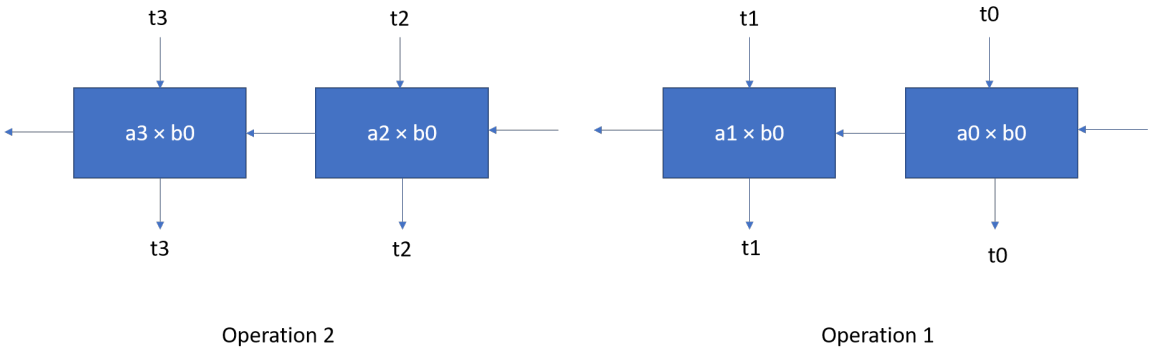

Figure 3.5: Dataflow in CIOS accelerator across iterations (Operations)

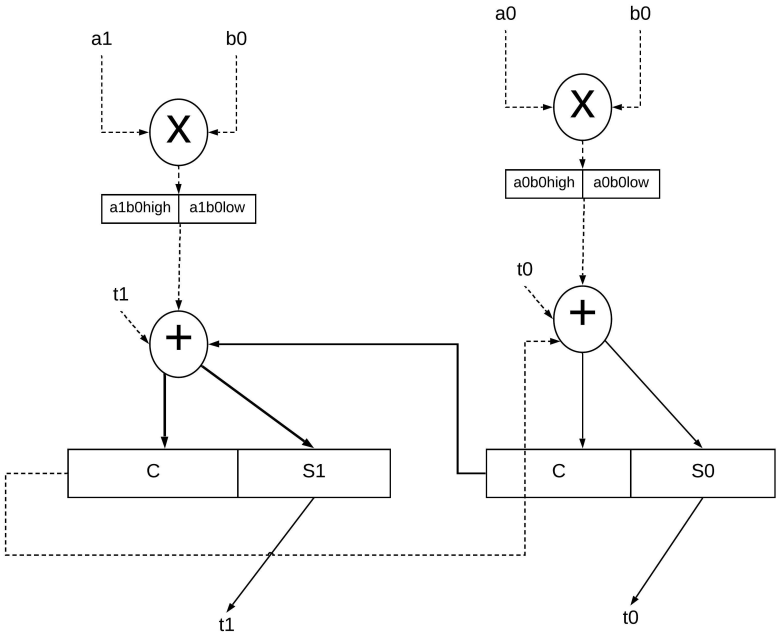

Figure 3.6: Architecture for CIOS accelerator

S0 write of 2 words of t. Result of one word (w *bits*) multiplications result in upto a two word (*2w bits*) long result. Addition of multiplication result to two w *bits* words can only be upto *2w bits* or two words long.

If the word size is assumed to be w, then the number of words for a multi-precision operand would be $1024/w = s$. To compute a single row of integer multiplication or montgomery reduction, the architecture has to be iterated $s/2$ times, since 2 blocks are computed at the same time. This has to be repeated s times to compute all rows required to compute Montgomery multiplication. The detailed analysis is shown in 3.1.

3.7 Finely Integrated Operand Scanning (FIOS) algorithm

The algorithm for the FIOS Montgomery multiplication is given in Algorithm 7. The algorithm obtains its name as it alternates between iterations of computing a block of integer multiplication followed by a block of montgomery reduction for a particular index.

Algorithm 7 FIOS Algorithm

1: **for** $i \leftarrow 0$ to $s-1$ **do**
2: $(C, S) \leftarrow t[0] + a[0] * b[i]$
3: $t[1] \leftarrow t[1] + C$
4: $m \leftarrow S * n'[0] \bmod W$
5: $(C, S) \leftarrow S + m * n[0]$
6: **for** $j \leftarrow 1$ to $s-1$ **do**
7: $(C, S) \leftarrow t[j] + a[j] * b[i] + C$
8: $t[j+1] \leftarrow t[j+1] + C$
9: $(C, S) \leftarrow S + m * n[j]$
10: $t[j-1] \leftarrow S$
11: **end for**
12: $(C, S) \leftarrow t[s] + C$
13: $t[s-1] \leftarrow S$
14: $t[s] \leftarrow t[s+1] + C$
15: $t[s+1] \leftarrow 0$
16: **end for**

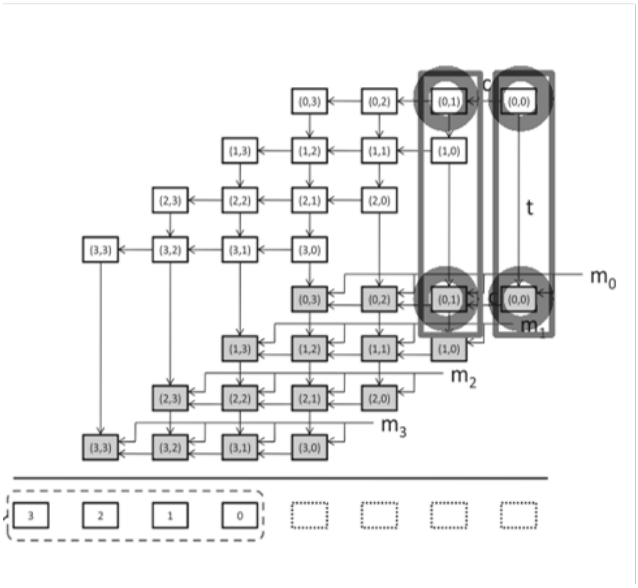

Figure 3.7: Block diagram representing dataflow of FIOS Montgomery multiplication

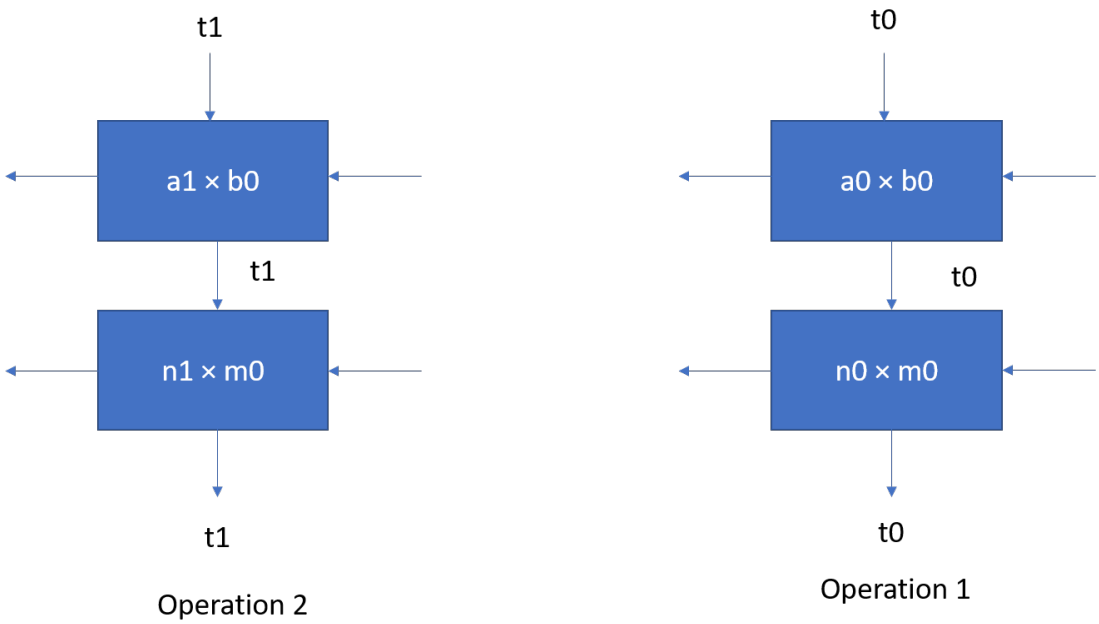

Figure 3.8: Dataflow in FIOS accelerator across iterations (Operations)

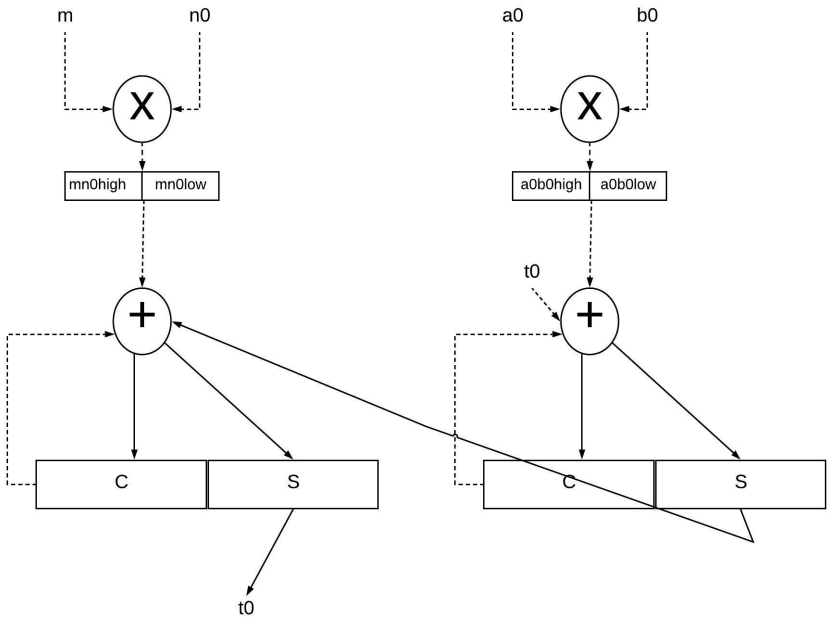

Figure 3.9: Architecture for FIOS accelerator

The minimal accelerator architecture to facilitate FIOS algorithm is given in figure 3.9. It consists of 2 multiplier units and associated adder units to compute partial products of integer multiplication and montgomery reduction of first index in parallel. Using block diagram representation of dataflow as shown in figure 3.7, one can see that using the architecture, computation can proceed in 2 block units at a time. For a particular index, the integer multiplication and montgomery reduction could be computed concurrently.

The inputs required for FIOS loop accelerator are 1 word of a, 1 word of b, 1 word of m, 1 word of n and 1 word of t as shown in figure 3.8. It produces 1 word of t as output. For the subsequent iteration/operation in the same row, the word b and n remain constant, but it requires new words of a, n and t.

The operations involved for one iteration are read of 1 word of a, 1 word of n, 1 word of t, 2 word-multiplications, 2 3-word additions, storing results in 4 registers *C1*,

Precision	Word size = 8 bit	Word size = 16 bit	Word size = 32 bit	Word size = w bit
Number of words	1024 / 8 = 128	1024 / 16 = 64	1024 / 32 = 32	1024 / w = s
Number of word-wise multiplications for Integer multiplication + Montgomery reduction	128 (2 8 bit * 8bit) blocks * 128 iterations	64 (2 16 bit * 16 bit) blocks * 64 iterations	32 (2 32 bit * 32 bit) blocks * 32 iterations	s (2 w bit * w bit) blocks * s iterations

Table 3.2: Number of word-wise multiplications for Montgomery multiplication using FIOS accelerator

Precision	Word size = 8 bit	Word size = 16 bit	Word size = 32 bit	Word size = w bit
Number of words	1024 / 8 = 128	1024 / 16 = 64	1024 / 32 = 32	1024 / w = s
Number of word-wise multiplications for Integer multiplication	64 (2 8 bit * 8bit) blocks * 128 iterations	32 (2 16 bit * 16 bit) blocks * 64 iterations	16 (2 32 bit * 32 bit) blocks * 32 iterations	s/2 (2 w bit * w bit) blocks * s iterations
Number of word-wise multiplications for Montgomery reduction	64 (2 8 bit * 8bit) blocks * 128 iterations	32 (2 16 bit * 16 bit) blocks * 64 iterations	16 (2 32 bit * 32 bit) blocks * 32 iterations	s/2 (2 w bit * w bit) blocks * s iterations

Table 3.3: Number of word-wise multiplications for Montgomery multiplication using proposed accelerator

S1, *C0*, *S0* write of 2 words of *t*. Result of one word (*w bits*) multiplications result in upto a two word (*2w bits*) long result. Addition of multiplication result to two *w bits* words can only be upto *2w bits* or two words long.

If the word size is assumed to be w, then the number of words for a multi-precision operand would be $1024/w = s$. Since a row of integer multiplication and montgomery reduction are computed in parallel the accelerator has to be iterated s times, since 2 blocks - one for integer multiplication and one for reduction need to be computed at the same time. This has to be repeated s times to compute all rows required to compute Montgomery multiplication. The detailed analysis is shown in table 3.2.

3.8 Proposed design

The algorithm proposed is given in Algorithm 5. Instead of alternating between iterations of computing a row of integer multiplication followed by a row of montgomery reduction, we compute 2 rows of integer multiplication or montgomery reduction at the same time.

Algorithm 8 Proposed Algorithm

```
1: for i ← 0 to s − 1 by 2 do
2:     C ← 0
3:     C1 ← 0
4:     for j ← 0 to s − 1 do                                  ▷ Inner Loop 1
5:         (C, S) ← t[j] + a[j] ∗ b[i] + C
6:         t[j] ← S
7:         (C1, S1) ← t[j + 1] + a[j] ∗ b[i + 1] + C1
8:         t[j + 1] ← S1
9:     end for
10:    (C, S) ← t[s] + C
11:    t[s] ← S
12:    t[s + 1] ← C
13:    (C1, S1) ← t[s + 1] + C1
14:    t[s + 1] ← S1
15:    t[s + 2] ← C1
16:    C, C1 ← 0
17:    m0 ← t[0] ∗ n′[0] mod W
18:    for j ← 0 to s − 1 do                                  ▷ Inner Loop 2
19:        (C, S) ← t[j] + m0 ∗ n[j] + C
20:        t[j] ← S
21:        if j = 0 then
22:            x ← t[1] + m0 ∗ n[1] + C
23:            m1 ← x ∗ n′[0] mod W
24:        end if
25:        (C1, S1) ← t[j + 1] + m1 ∗ n[j] + C1
26:        t[j + 1] ← S1
27:    end for
28:    (C, S) ← t[s] + C
29:    t[s] ← S
30:    t[s + 1] ← t[s + 1] + C
31:    (C1, S1) ← t[s + 1] + C1
32:    t[s + 1] ← S1
33:    t[s + 2] ← t[s + 2] + C1
34:    for j ← 0 to s + 2 do
35:        t[j] ← t[j + 2]
36:    end for
37: end for
```

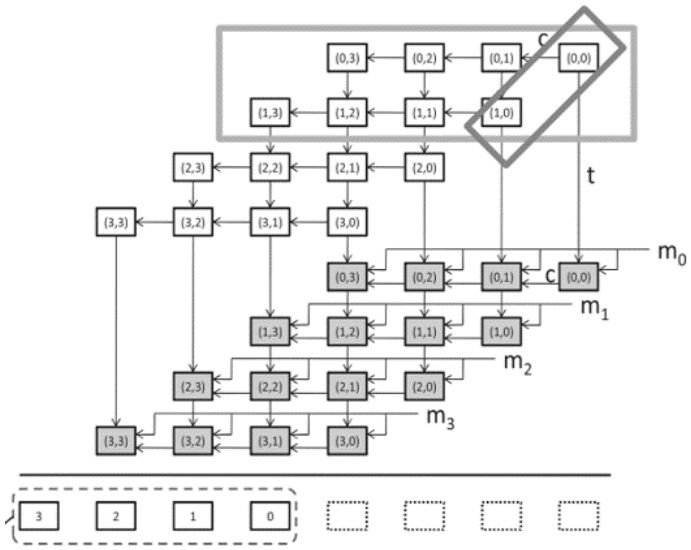

Figure 3.10: Block diagram representing dataflow of proposed algorithm for Montgomery multiplication

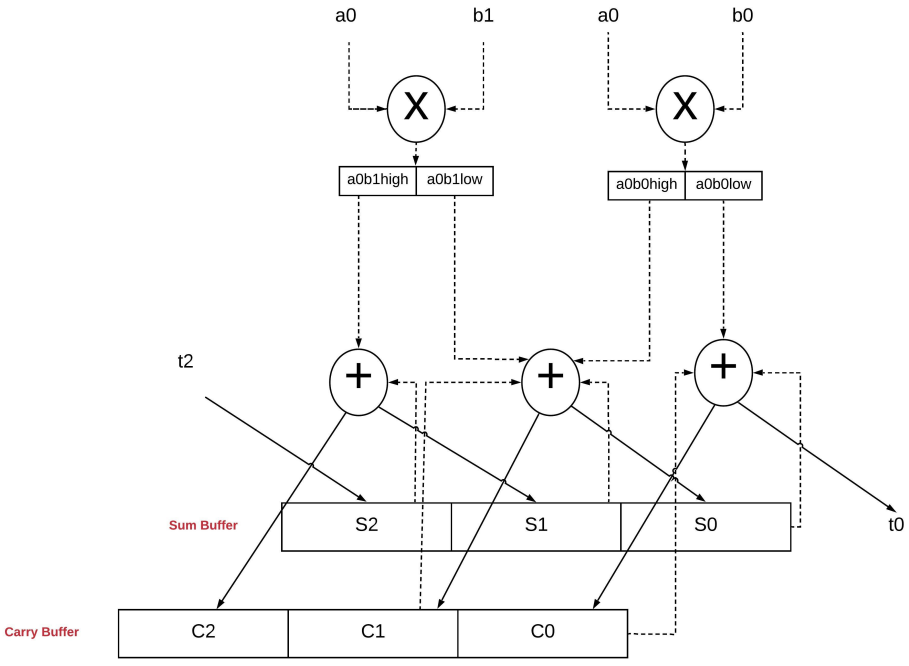

Figure 3.11: Architecture for proposed accelerator

The minimal accelerator architecture to facilitate the proposed algorithm is given in figure 3.11. It consists of 2 multiplier units and associated adder units to compute partial products of first two indices across 2 rows in parallel. Using block diagram representation of dataflow as shown in figure 3.10, one can see that using the architecture, computation can proceed in 2 block units at a time. If integer multiplication (or montgomery reduction) is being computed, the 2 blocks compute integer multiplication (or montgomery reduction) results for consecutive indexes in a concurrent manner.

The inputs required for the proposed accelerator are 1 word of a, 2 words of b and 2 words of t in Algorithm 8 and from figure 3.10. It produces 1 word of t as output. For the subsequent iteration/operation in the same row, the words b remains constant, but it requires 1 new word of a, and 1 new word of t.

The operations involved for one iteration are read of 1 word of a, 1 word of t, 2 word-multiplications, 2 3-word additions, 1 4-word addition storing results in 6 registers *C2*, *S2*, *C1*, *S1*, *C0*, *S0* and write of 1 word of t. Result of one word (w *bits*) multiplications result in upto a two word (*2w bits*) long result. Addition of the upto 4 words cannot exceed *2w bits*.

If the word size is assumed to be w, then the number of words for a multi-precision operand would be $1024/w = s$. To compute 2 row of integer multiplication or montgomery reduction, the architecture has to be iterated s times, since 2 blocks are computed at the same time. This has to be repeated *s/2* times to compute all rows required to compute Montgomery multiplication. The detailed analysis is shown in 3.3.

We observe that in the loop function for montgomery reduction, the first iteration with loop iterator j as 0, the value of m1 is determined after calculation of t[0] and C. This is because the second row of the Montgomery reduction requires the value m1, which can be calculated only after the lower 2 blocks of the first row are calculated. In

order to transfer the inputs required to calculate m1 to the processor, if the accelerator has to be used, then the accelerator, could be run for 2 iterations by setting m1 value to be 0, so that x can be calculated and transferred to the processor. If the number of iterations of the accelerator block is s/2 * s iterations, then the additional computation needs to be done for 2 * s/2 which is s iterations, thus increasing the total iterations by s.

The discussion of accelerator architectures to be analyzed is complete. In the next chapter the experimental setup, results and observations made while comparing the architectures are discussed.

CHAPTER 4

EXPERIMENTS, ANALYSIS AND RESULTS

4.1 Design space exploration of accelerator using high level synthesis

High level synthesis facilitates effective design space exploration of hardware architectures and helps in improving design productivity for hardware designers. High level synthesis is an automated design process that aids in transforming high level specification usually implemented in C/C++ or SystemC to optimized hardware solutions. It provides the opportunity for a designer to consider various architectural solutions without altering the high level specification. The main idea behind HLS is to generate different RTL implementations by raising the abstraction level of the design process. By setting the optimization directives (also known as knobs) provided by HLS, a designer can explore alternative RTL implementation for the same high level specification. Some standard knobs provided by HLS tools are given in table 4.1.

Knob	Settings and Effects
Loop Manipulations	Unrolling: Replicates operations in loop body Pipelining: Pipelines the operations in the loop body Breaking: Inserts additional states in the loop body.
Array Mappings	Maps the arrays to registers or on-chip memories
Clock Period	Sets the target clock period for the synthesis

Table 4.1: Standard knobs provided by HLS tools

High level synthesis enables the transformation of a behavioral description of an algorithm with no timing information into a hardware implementation in RTL that

is cycle accurate. HLS uses three processes to enable this - Scheduling, resource allocation and resource binding. Scheduling is used to assign each operation to a time step which correspond to a particular clock cycle. Resource allocation is the process which determines the number and type of hardware modules to be used to implement the desired operation. Resource binding is the process that assigns operations to the hardware modules as determined by the resource allocation process.

Intel HLS Compiler and Xilinx Vivado HLS are HLS tools provided by the two largest FPGA vendors. The high level languages do not have the concept of system clock. In this work, we use Vivado HLS to conduct design space exploration.

4.2 Testbench for experiments

We conduct our experiments using Vivado HLS tool provided by Xilinx. Xilinx Vivado HLS tranforms a high level specification, written in C, C++, SystemC or an OpenCL API C Kernal into a RTL implementation that can be synthesized into a Xilinx Field Programmable Gate Array (FPGA). In applications where the FPGA is used to verify an ASIC design, HLS would be a good tool and the same prototyping flow can be used. However, since HLS will use built-in resources such as DSP48 to infer the RTL, these cannot be used for the ultimate ASIC design, as the FPGA specific design components will not translate.

Vivado HLS design flow can be summarized as follows:

1. Compile, Execute and Debug C specification.

   The first step is to compile and execute the C specification to make sure the results match with the golden reference file. A C testbench file is responsible for calling the C specification implemented as a C function, retrieve the results and compare against the golden reference file. The designer can make use of the debugging functionality provided by Xilinx Vivado to resolve any issues at this stage.

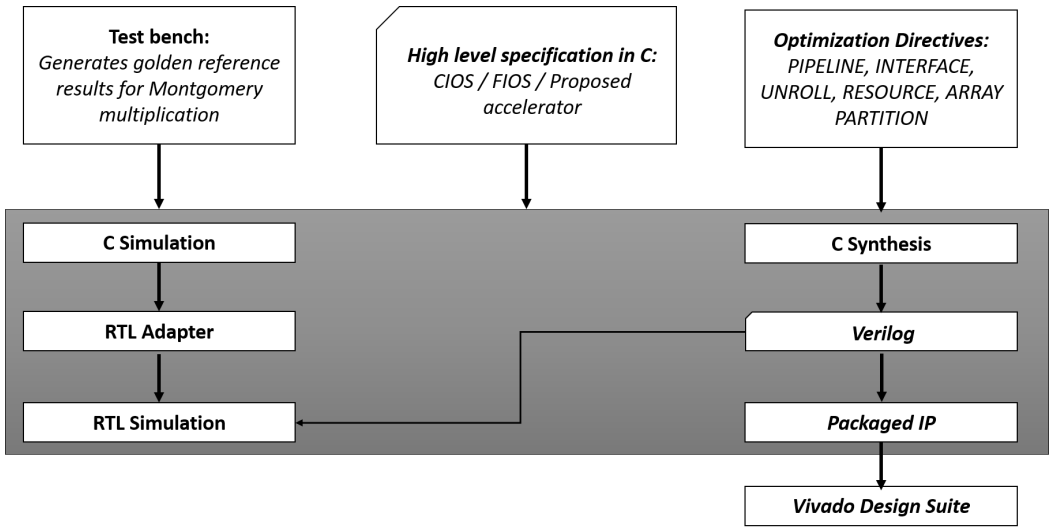

Figure 4.1: Vivado HLS Design Flow

2. Synthesize C specification into an RTL implementation.

   Optimization directives and constraints can be applied to the design to synthesize the C specification into RTL implementation for the desired FPGA.

3. Generate reports regarding hardware resource utilization and timing.

   These reports can be utilized to explore the design space and modify the optimization directives and constraints until the desired design criteria is achieved.

4. C/RTL Cosimulation.

   The C testbench added for the C simulation can be used to verify the functionality of the RTL.

5. Package RTL implementation into an IP.

   The RTL can be exported into an IP in any of the following Xilinx IP formats: Vivado IP Catalog, System Generator for DSP, Synthesized Checkpoint.

4.2.1 Opimization directives

Optimization directives are features provided by Vivado to produce a micro-architecture that satisfy the desired performance and design goals.

The optimization directives provided by Vivado are allocation, array_map, array_partition, array_reshape, clock, data_pack, dataflow, dependence, expression_balance, function_instantiate, inline, interface, latency, loop_flatten, loop_merge, loop_tripcount, occurence, pipeline, protocol, reset, resource, stream, top, unroll.

The optimization directives used in this work are discussed in detail below:

Loop Pipelining: Loops in the high level specification can be pipelined when implemented in the hardware to accept new inputs every N clock cycles, where N is the loop initiation interval. Pipelining of designs necessitates use of registers between pipeline stages generating high latency, high throughput designs. High throughput is possible if the loop iterations can be overlapped, which might be restricted if there are dependencies across iterations.

Loop Unrolling : Loops in the high level specification can be unrolled to exploit spatial parallelism, to reduce the number of iterations to N/F, where N is the total number of iteration and F is the unroll factor. To enable this, there must be F copies of the loop operation.

Reset : This directive is used to add or remove reset on a specific variable. When the reset signal is applied to the accelerator or IP block, this directive helps control the registers which should be reset. Fine grain control over reset becomes useful when static or global arrays or variables are present in the design.

Interface : This directive helps specify how the RTL ports need to be created from the function description.

Resource : This directive is used to specify a specific library resource (core) to implement a variable (array, arithmetic operation or function argument) in the RTL.

Array Partitioning : This directive can be used to restructure arrays synthesized in the design into individual elements.

4.2.2 Design analysis key concepts

Area : Vivado HLS generates a utilization estimate report that provides the number of BRAMs, DSPs, FFs and LUTs required to implement the design. We compare area of different designs by comparing the number of LUTs that would be required to implement the design.

Latency : Vivado HLS generates timing reports that provides the number of clock cycles required by a particular design to complete an operation for a particular clock period. We synthesize the designs for clock periods - 5ns, 10ns, 15ns, 50ns. These were the typical frequencies used in other works on accelerators for sensor nodes designed in reconfigurable hardware. **Data transfer rate** : It is determined as the ratio of the total number of words transferred to and from the accelerator to the total number of clock cycles taken to complete accelerator operation.

4.3 Experimental Setup

Xilinx Vivado HLS was used to simulate and synthesize the designs by following the Vivado HLS design flow as explained in section 4.2. The high level specification of the architectures presented in the previous chapter were given as design specification to Vivado HLS. Various optimization directives were applied to the design to perform analysis. The high level designs were synthesized for Zynq device in Zedboard - xc7z020clg484-1. The total number of LUTs, FFs, DSP48Es and BRAM_18Ks available in the device are 53200, 106400, 220 and 280 respectively. The aim of the experiments are:

- To implement the proposed accelerator architecture and verify its functionality.
- To implement accelerators suitable for FIOS and CIOS algorithms and compare them with the proposed architecture

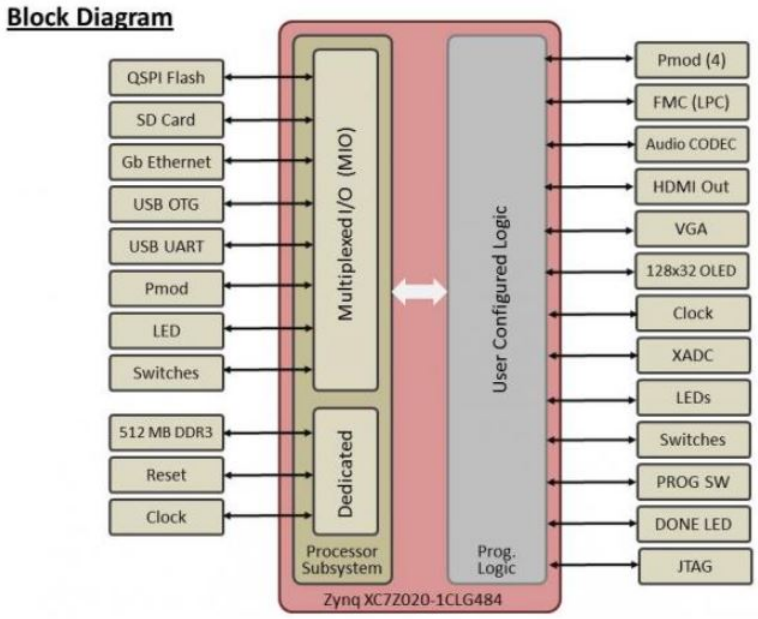

Figure 4.2: Zedboard-Zynq block diagram [4]

- To analyze the design space by making use of optimization directives provided by HLS.

4.3.1 Number of Ports

The architecture hardware blocks or accelerators are assumed to have only specific number of ports to transfer data in and out of the system as discussed in the previous chapter. They are tabulated below in table 4.2.

	FIOS	**CIOS**	**Proposed ACC**
Ports 2		Single unidirectional port for a, Single bidirectional port for t	Single unidirectional port for a, Single bidirectional port for t
Ports 3	Single unidirectional port for a, Single unidirectional port for n, Single bidirectional port for t	Single unidirectional port for a, Dual bidirectional port for t	Single unidirectional port for a, Dual bidirectional port for t
Ports 4	Single unidirectional port for a, Single unidirectional port for n, Dual bidirectional port for t	Dual unidirectional port for a Dual bidirectional port for t	Dual unidirectional port for a, Dual bidirectional port for t
Unconstrained	No restriction for data transfer	No restriction for data transfer	No restriction for data transfer

Table 4.2: Port definition and usage for FIOS, CIOS and proposed accelerator architecture

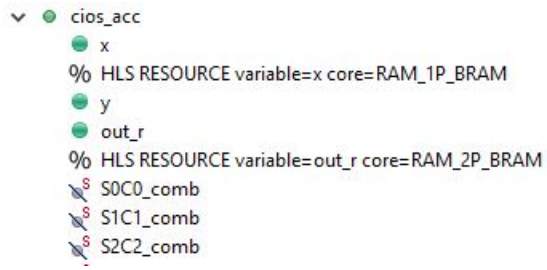

Figure 4.3: Resource directive applied to operands

To facilitate this, we assume that the inputs to the accelerator are stored in RAMs (Random Access Memory) external to the accelerator. Vivado HLS provides an optimization directive under Resource where the inputs can be chosen to be stored in RAM_1P_BRAM (Single Port) or RAM_2P_BRAM (Dual Port). Block RAMs (or BRAM) stands for Block Random Access Memory. Block RAMs are used for storing data in Xilinx FPGA. We also intend to compare the design if there is no restriction to input availability. To facilitate unconstrained availability of inputs to the accelerator, the optimization directive 'Array Partition complete' is applied to the operands.

Note that the FIOS design cannot be synthesized with 2 ports as it requires minimum of 3 ports to transfer inputs.

4.3.2 Word size

The word size, denoted by w of the word wise multiplication algorithms were varied to be of 8bits, 16 bits and 32 bits to study the variation in their performance. The architecture demands word size and double word size registers, The registers were defined to be of different data types according to the word size requirement as shown in the table 4.3. High level specification had to be modified to implement this variation, see listing 4.1.

```
1 #ifndef CIOS_H_
2 #define CIOS_H_
3
4 #define MPM_LEN_IN_BITS 1024
5 #define WORD_LEN_IN_BITS 16 //int = 32 bits
6 #define N MPM_LEN_IN_BITS/WORD_LEN_IN_BITS
```

	8 bit	16 bit	32 bit
word	char	short	int
double word	short	int	long long

Table 4.3: Data types in HLS for register word sizes

```
7  #define DISP_WORDS_PER_ROW 8
8
9
10 /*
11 //Uncomment if word size = 8 bits
12 typedef unsigned char word;
13 typedef unsigned short double_word;
14 */
15
16
17 //Uncomment if word size = 16 bits
18 typedef unsigned short word;
19 typedef unsigned int double_word;
20
21
22 /*
23 //Uncomment if word size = 32 bits
24 typedef unsigned int word;
25 typedef unsigned long long double_word;
26 */
27 void cios(word x[N],word y,word out_r[N+1]);
28
29 #endif
```

Listing 4.1: C code snippet that shows word size variants

4.3.3 Other design points

Vivado HLS allows the designs to be synthesized with different clock periods. The clock periods were varied to be 5ns, 10ns, 15ns and 50ns to observe the variation in the synthesized design. Vivado HLS also provides an optimization directive called PIPELINE that could be applied to loops. If the loops are capable of being pipelined, this would give to designs with better throughput.

4.4 Experimental results

Once the designs and simulated and synthesized, Vivado HLS provides utilization and timing reports for design analysis. The first analysis performed, is to compare both pipelined and unpipelined versions of all architectures for different port availability. The comparison is done in terms of the number of clock cycles taken to achieve Montgomery Multiplication and the area requirement in terms of LUTs for each design.

The architectures for CIOS accelerator, FIOS accelerator and proposed accelerator and their pipelined variations are simulated and synthesized with a clock period of 5ns, 10ns, 15ns and 50ns. The results for clock period of 5ns are plotted here. The designs are also synthesized for varying port availability. In the first set of graphs indicated by Ports vs Clock Cycles, the number of clock cycles taken to compute Montgomery multiplication is plotted against the number of ports available to the accelerator. This is plotted for various word sizes in figures 4.4, 4.8, 4.12. The same set of graphs are plotted with latency in microseconds instead of clock cycles in 4.6, 4.10, 4.14. For all the base designs, the latency does not seem to vary much with increase in the number of ports. For the pipelined versions however, as the accelerator contains more ports, the performance of the systems seems to improve.

In the second set of graphs indicated by Ports vs LUTs, the number of LUTs to be utilized for the design is plotted against the number of ports available to the accelerator. This is plotted for various word sizes in figures 4.5, 4.9, 4.13.

Since the area of the pipelined version of proposed accelerator is higher than the rest of the architectures, we unroll the CIOS, FIOS, both base and pipelined versions by unroll factor of 2, 4 and 8 to see if increasing the area leads to better performance. We observe that even after increasing the area, the latency does not as compared to proposed accelerator architecture. This is plotted in figures 4.7, 4.11, 4.15.

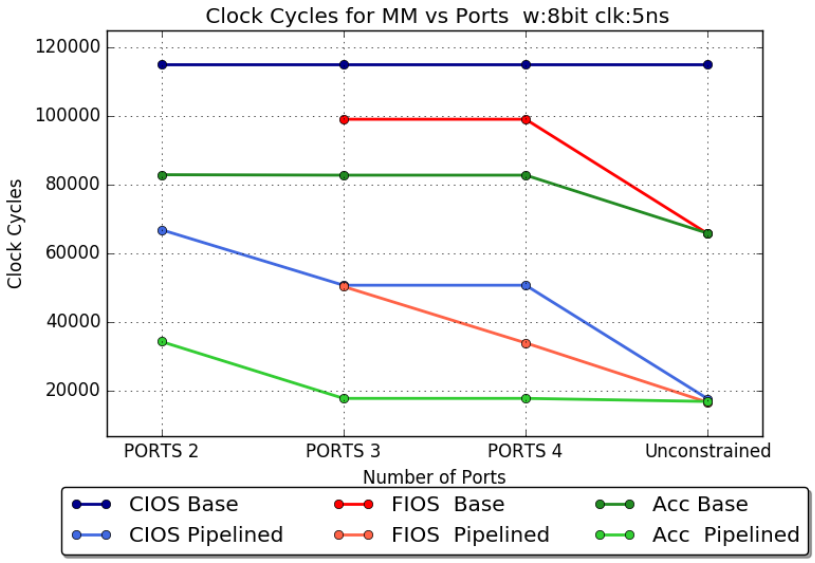

Figure 4.4: Number of ports vs number of clock cycles to calculate MM(us) for accelerator with 8 bit word size

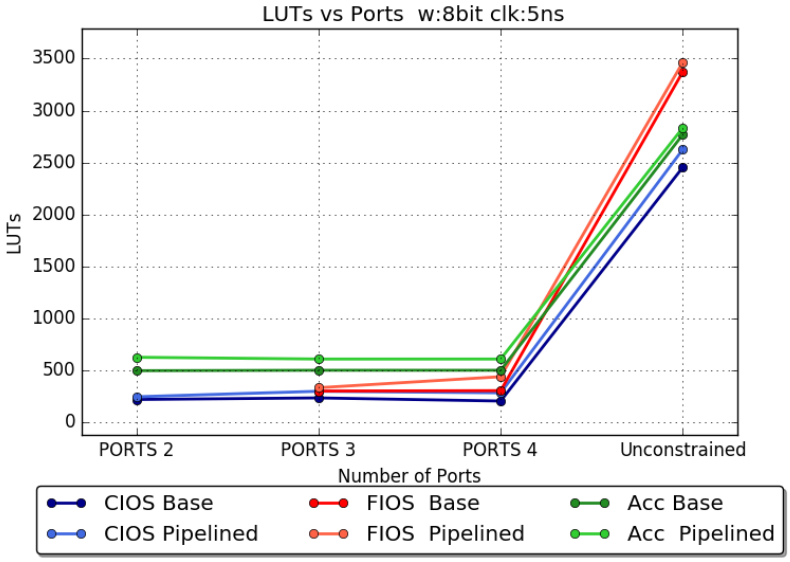

Figure 4.5: Number of ports vs LUTs utilized in accelerator with 8 bit word size

4.4.1 Latency calculation

For all algorithms, a sequential architecture required to implement the algorithm is synthesized. The architecture as such is capable of implementing only 2 blocks

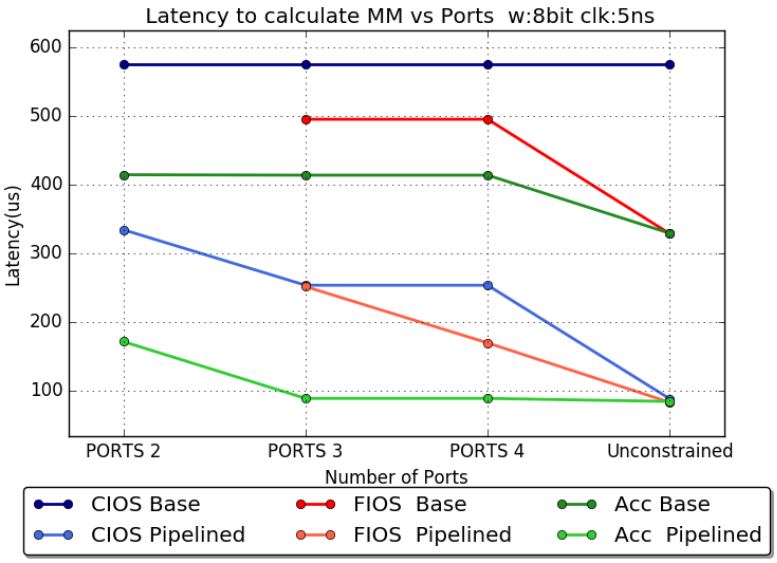

Figure 4.6: Number of ports vs latency to calculate MM(us) for accelerator with 8 bit word size

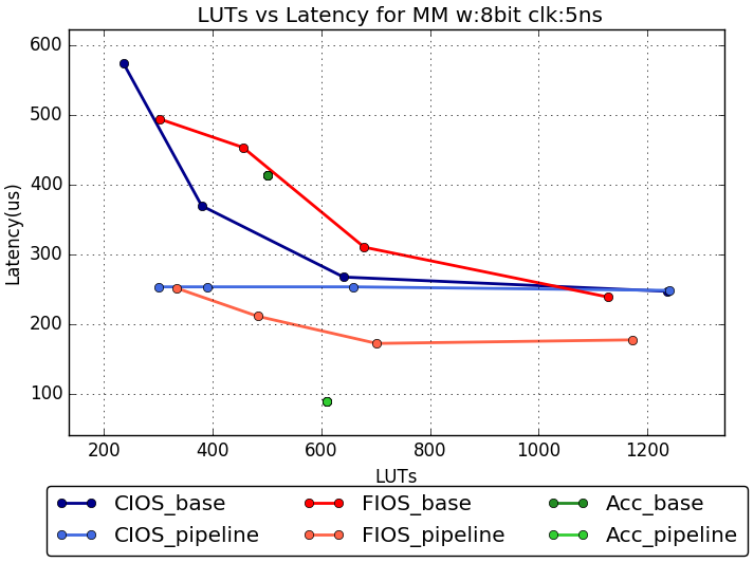

Figure 4.7: LUTs utilized vs Latency to calculate MM(us) for designs with 8 bit word size

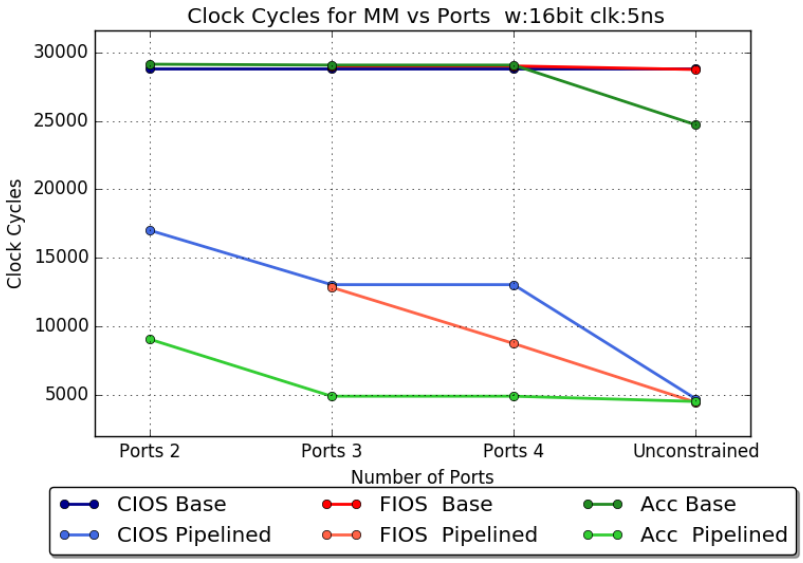

Figure 4.8: Number of ports vs number of clock cycles to calculate MM(us) for accelerator with 16 bit word size

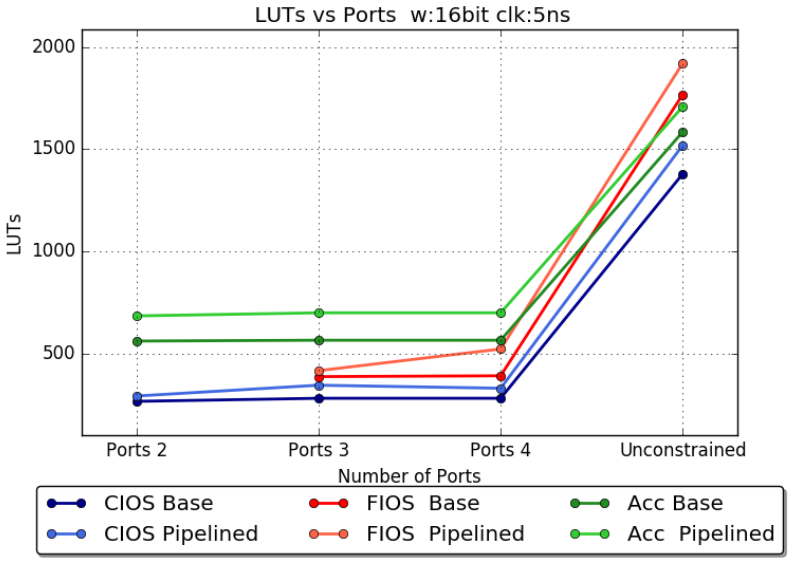

Figure 4.9: Number of ports vs LUTs utilized in accelerator with 16 bit word size

in all cases as discussed previously. They have to be iterated a specific number of times to implement the Montgomery multiplication. Hence the latency is calculated

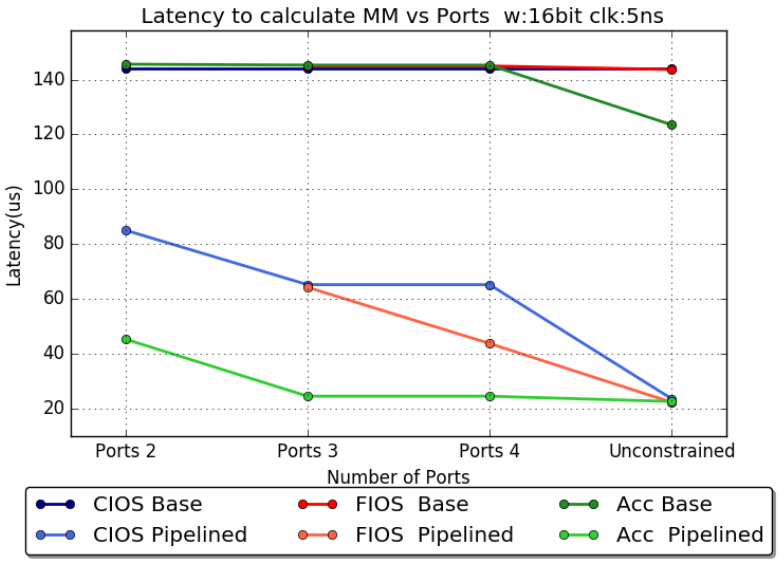

Figure 4.10: Number of ports vs latency to calculate MM(us) for accelerator with 16 bit word size

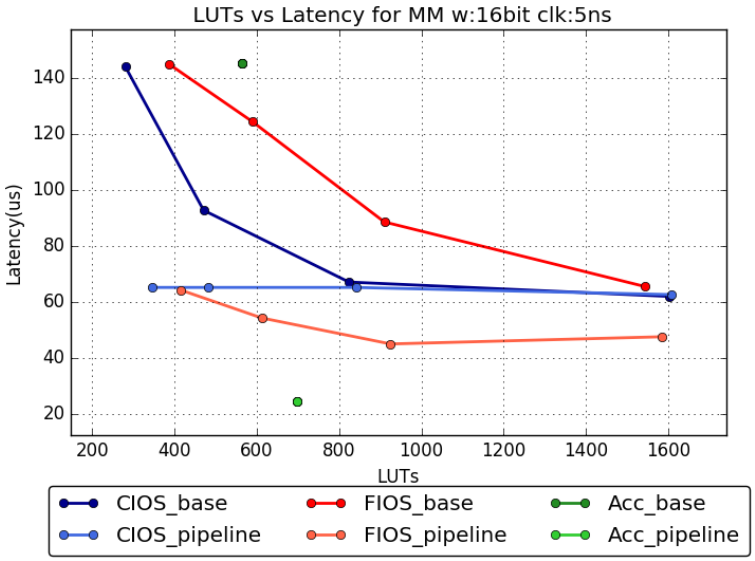

Figure 4.11: LUTs utilized vs Latency to calculate MM(us) for designs with 16 bit word size

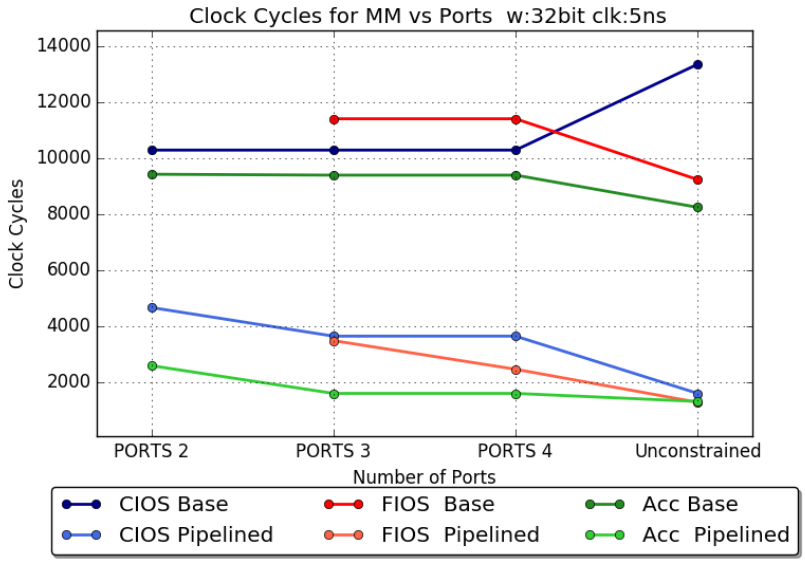

Figure 4.12: Number of ports vs number of clock cycles to calculate MM(us) for accelerator with 32 bit word size

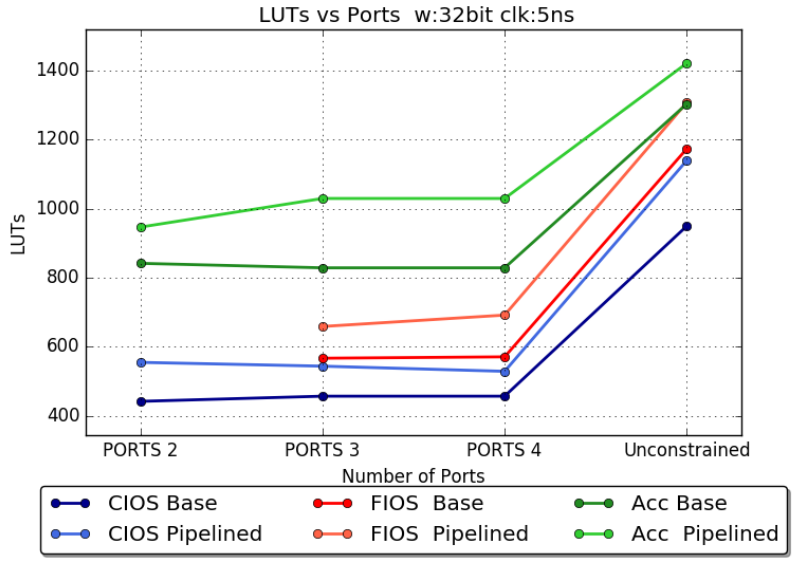

Figure 4.13: Number of ports vs LUTs utilized in accelerator with 32 bit word size

by finding out the time taken by each architecture to implement a row in the data

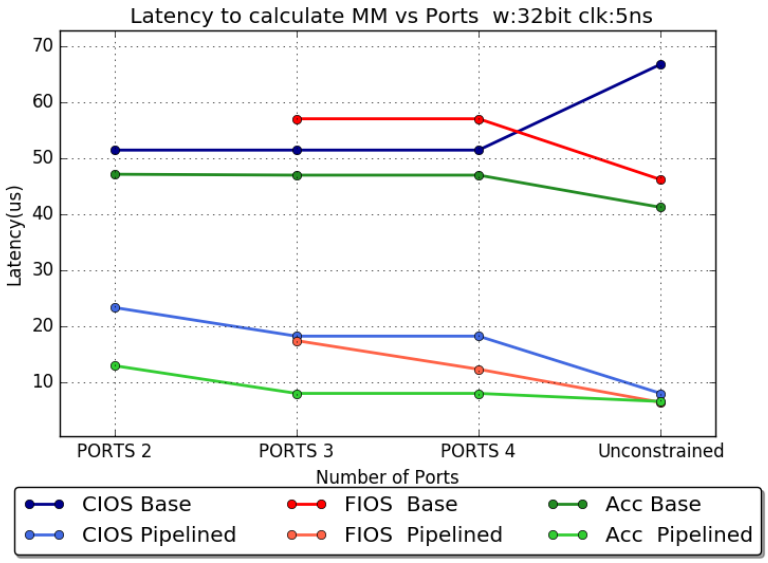

Figure 4.14: Number of ports vs latency to calculate MM(us) for accelerator with 32 bit word size

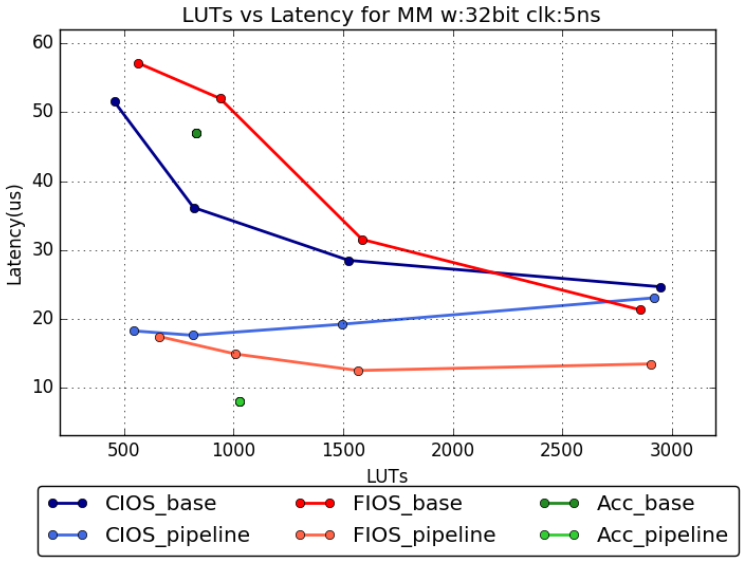

Figure 4.15: LUTs utilized vs Latency to calculate MM(us) for designs with 32 bit word size

flow block diagram and then extrapolating it to implement complete Montgomery Multiplication for valid comparison.

If the multiprecision number is split into s words of w bits each, then the time taken to compute Montgomery Multiplication can be computed according to the following equations. Since calculation of m is not part of hotspot function, it is not implemented in the accelerator architecture. Hence, the time taken to compute m is not accounted for.

Using CIOS accelerator, if $Latency_{row}$ is the time taken to complete a row of operations in the data flow block diagram, then the latency to complete Montgomery multiplication would be,

$Latency_{MM} = Latency_{row} * s * 2$

This is because there are s rows to compute integer multiplication and s rows to compute Montgomery reduction.

Using FIOS accelerator, if $Latency_{row}$ is the time taken to complete a row in the data flow block diagram, then the latency to complete Montgomery multiplication would be,

$Latency_{MM} = Latency_{row} * s$

This is because a block of integer multiplication and Montgomery reduction is computed in an interleaved manner. Since the blocks are computed concurrently, there are only s rows in total to compute both integer multiplication and Montgomery reduction.

Using the proposed design for accelerator, if $Latency_{row}$ is the time taken to complete a row in the data flow block diagram, then the latency to complete Montgomery multiplication would be,

$Latency_{MM} = Latency_{row} * s$

This is because 2 blocks of integer multiplication or montgomery reductions are computed in parallel. Hence, there are only s rows in total to compute integer multiplication and Montgomery reduction.

4.5 Observation and analysis

Let iteration 1 be defined as the time taken to generate the first set of partial products by running the accelerator once. The accelerator design proposed generates partial products for indices 0, 1 and 2 in the first iteration. In the second iteration it computes partial products for indices 1, 2 and 3. In the first iteration, the architecture is expected to produce final product for index 0. The rest of the partial products are registered for the use in the following iterations. In the second iteration, the final product for index 1 is expected. The inputs required to generate the final product is generated in the current iteration and the previous iteration which have been registered. In order to produce output final product for a specific iteration, the datapath involves a single block (a single multiplier and associated adder circuitry).

For the accelerator that implements CIOS architecture, the first iteration deals with indices 0, 1 and 2. The second iteration deals with computation of outputs for indices 2, 3 and 4. This necessitates that after the first iteration, the final products for indices 0 and 1 should be computed. After the second iteration, the final products for indices 2 and 3 should be computed. In order to do so, the datapath of the final product for the higher index involves 2 blocks (2 multipliers and associated adder circuitry).

For the accelerator that implements FIOS architecture, the first iteration deals with computation of partial products for indices 0 and 1. The second iteration deals with computation of outputs for indices 1 and 2. This necessitates that after the first iteration, the final product of index 1 should be computed. In order to do so the

datapath of the final product for a specific iteration involves 2 blocks (2 multipliers and associated adder circuitry).

We observe that the pipelined versions of the accelerators performs better than the corresponding base designs for all architectures. This could be because there is scope for some units of the design to be operated in parallel. The adder operation of one iteration and the multiplier operation of the subsequent iteration do not have data dependency and hence can be operated in parallel.

Since there is no data dependency between the 2 multiplier blocks or the 2 adder blocks to produce output for that iteration in the proposed accelerator architecture, the various units in the architecture is much more suited for pipelining. However, for CIOS and FIOS architectures, since the datapath involves either the multiplier or the adder unit to be stalled by the other unit, there is a data dependency between various units, and hence the accelerator architecture cannot be completely pipelined. Hence the performance of the pipelined version of the proposed accelerator is better than the other designs, but since it involves inputs to be registered and additional adder circuitry, the design consumes more area as can be seen from the number of LUTs consumed on the device.

4.5.1 Effect of increasing number of ports on each design

We study the effect of increasing the number of ports in the accelerator architectures on the performance for all word lengths. We observe that as the number of ports increases, there is an improvement in the performance for all pipelined versions of the accelerators. However, their performance tend to be comparable when the input availability is unconstrained. This proves that the architecture of the accelerator design impacts the performance of the accelerators depending on input availability.

4.5.2 Effect of increasing area in other architectures

We observe that the latency of the pipelined version of the proposed accelerator is lower but at the cost of higher area. In order to see if the performance of the reference architectures improve with increase in area, we unroll the design by factors of 2, 4 and 8 and compare the LUTs consumed and latency. In the set of graphs indicated by LUTs vs Latency, this information is plotted. We observe that even with increase in area, the other designs cannot achieve the performance seen in the proposed design.

4.5.3 Effect of word size in the performance of proposed accelerator design

The speedup of the proposed accelerator design against the design CIOS pipelined for when accelerator has 2 ports and FIOS pipelined when accelerator has 3 and 4 ports are plotted in table

	Ports 2	**Ports 3**	**Ports 4**
8 bit	47.55	64.63	48.65
16 bit	46.6	61.7	43.8
32 bit	44.5	54.2	35.1

Table 4.4: Speedup of proposed design

	Ports 2	**Ports 3**	**Ports 4**
8 bit	2.54	1.82	1.38
16 bit	2.34	1.68	1.33
32 bit	1.71	1.56	1.49

Table 4.5: Area requirement of proposed design as compared to base design

We observe that as word size increases, the speedup observed decreases. This is because as w goes higher, the number of words (s) to which the multi-precision number gets split into decreases. As the number of words decreases, the number of iterations of the accelerator to calculate Montgomery product also decreases, thus reducing the effectiveness of the accelerator in speeding up the overall algorithm.

4.6 Implementation in ZedBoard

We prototype the 32 bit word size version of our proposed accelerator with interfaces to single port BRAMs in the Zedboard.

Designs generated in Vivado HLS could be exported as a Vivado HLS IP (Intellectual Property) to provide easy integration of IP with other development tools in Vivado and ISE design suite.

The IP can be exported in IP-XACT format, or as a pcore for XPS-based (Xilinx Platform Studio based) system or for use in system generators for DSPs. IP-XACT is an xml format that defines and describes IP to facilitate their use in Integrated Circuits. Exporting in IP-XACT format allows module to be easily integrated to Vivado IP Integrator design and hence we adopt this method.

Before exporting the design as an IP, we assign block-level protocol for the design to be saxilite (AXI4-Lite). AXI is a protocol belonging to ARM AMBA family of microcontroller buses. AMBA is an on-chip interconnect specification, allowing connection and management of peripherals and controllers in a multi-master design. The AXI protocol is optimized for Xilinx FPGA implementations and is used as the means of communication between IP cores of an FPGA design. There are 3 variations of the AXI4 interface - AXI4, AXI4-Lite and AXI4-Stream. We use AXI4-Lite interface for the IP block interface. We specify the operand interfaces to be 'bram' and specify all of them to be of single port.

Using Vivado 2018.3 IDE, we integrate the exported IP "cios_acc_0" into Zynq SoC device of Zedboard. In the SoC, we configure the Zynq processing system (PS) and instantiate other IPs like "cios_acc_0" and BRAM IPs for each of the operand in the (Programmable Logic) PL part. It is not enough to just integrate the the exported IP "cios_acc_0" to BRAMs. The Zynq processor should also be able to communicate to the BRAMs to read and write values to it. For this purpose we make all the BRAMs to be dual port and include AXI BRAM controllers for Zynq PS to

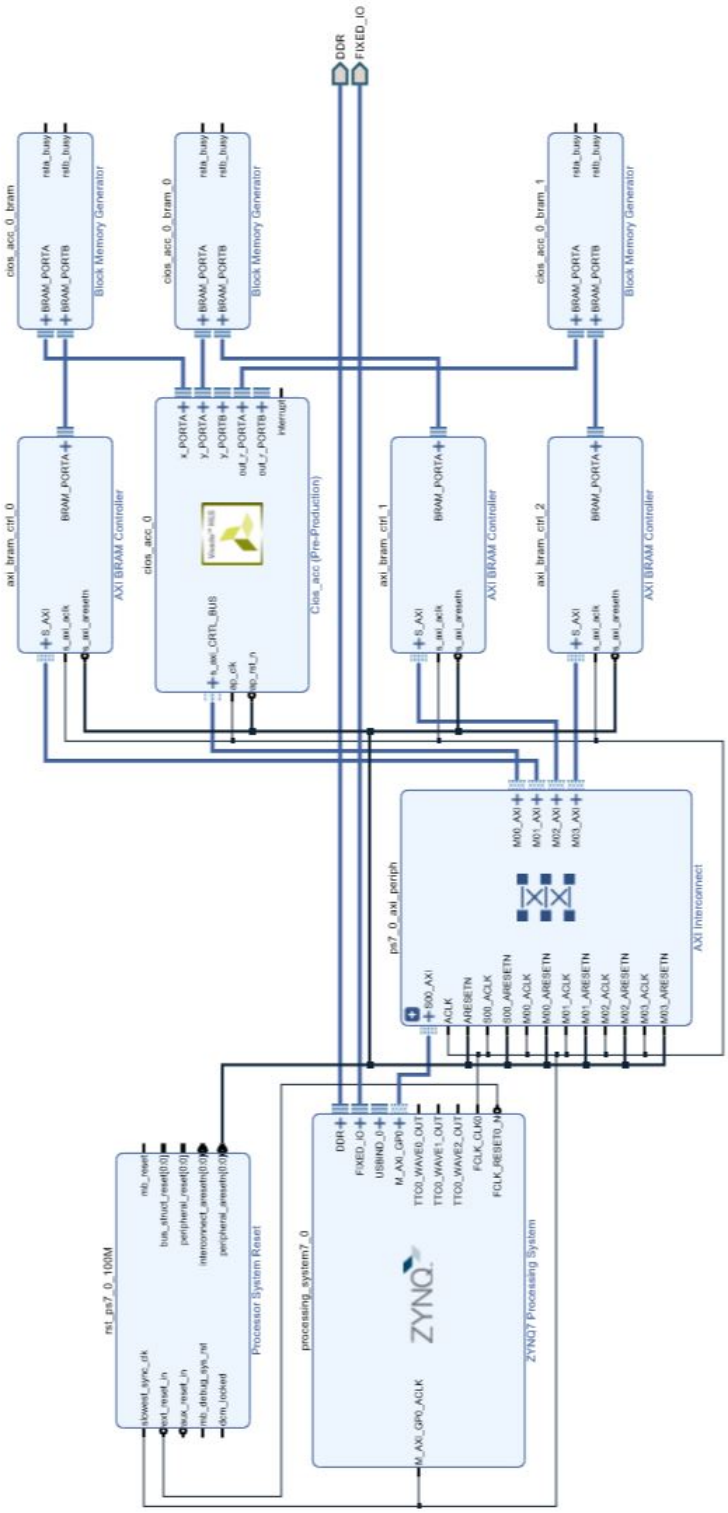

Figure 4.16: SoC design with accelerator synthesized in Zedboard

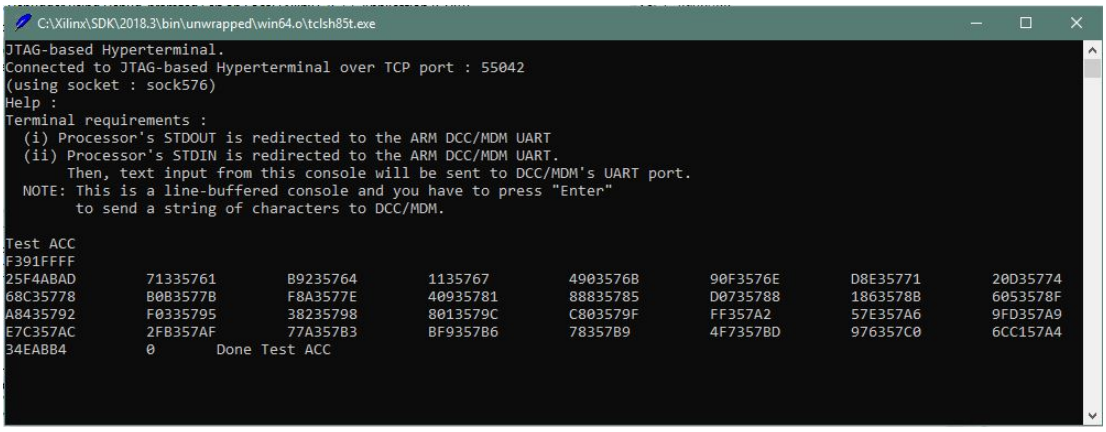

Figure 4.17: Interfacing with Zedboard via JTAG terminal

interact with the BRAMs. One port is used to interface with our IP and the other port is used to interface with Zynq PS. Vivado IP integrator provides *Connection Automation* and Block Automation to aid in connections involvig AXI interfaces, clk and reset signals. The design synthesized in the Zedboard is shown in figure 4.16.

The design is then validated, a HDL wrapper is created and the bitstream to program FPGA is generated. The Xilinx SDK 2018.3 helps in programming the FPGA with the bitstream and developing the software for Zynq Processor System. The code snippet is shown in Listing B.1. The cios_acc_0 IP is first initialized and sanity checks are performed. The x, y and out BRAMs are initialized with input data. The cios_acc_0 is then invoked to compute the results. The Zedboard is connected to the PC COM6 Serial Port with baud rate of 115200. After booting the system from SD Card, the PC is interfaced to the Zedboard via JTAG port. The results of the accelerator are verified through JTAG terminal as shown in figure 4.17.

Thus, the proposed accelerator is compared with accelerator architectures implementing CIOS and FIOS algorithms in terms of performance and area. The functionality of the proposed accelerator architecture is verified and is prototyped by designing a Zynq SoC in Xilinx Zedboard.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The RSA algorithm, one of the most prevalent PKC schemes, was profiled to determine the bottleneck function which could be accelerated to achieve performance improvement in resource constrained edge devices. The bottleneck function was determined to be the *mpi_mul_hlp* function, which is a loop function of the Montgomery multiplication algorithm, commonly used for modular multiplications in the RSA key generation algorithm. A novel architecture for accelerating the loop function of the Montgomery multiplication was proposed, which was designed with system level constraints in mind. The design was simulated, synthesized and compared against other accelerator architectures that implement variants of Montgomery algorithm in the literature. We observe that the pipelined version of the proposed accelerator performs better than other architectures when the number of ports in the accelerators are 2, 3 and 4. The average speedups observed are 46.21%, 60.27%, 38.1% as compared to the base designs when the number of ports available in the system are 2, 3 and 4 respectively with area requirement 2.19, 1.68 and 1.4 times higher than the base designs to which are used for comparison.

5.2 Future Work

In this work, only the operand scanning methods to compute Montgomery multiplication have only been analyzed due to their uniform structure. Minimal architectures in terms of area and capable of computing Montgomery product in a sequential

manner for product and hybrid scanning methods could be synthesized and a comparison could be established in terms of area and performance. Another possible extension would be to measure the power and energy consumed by the various accelerator architectures that implements Montgomery multiplication in a sequential manner and establishing a study based on power profile. As both power and performance increase linearly, the energy is expected to remain the same (for the multiplier). In all of these architectures, we assumed that the input operands were stored in an external memory and fed to the accelerator as and when required. In most cases, the same word has to be fed a number of times to complete montgomery multiplication. By assuming that the accelerator has internal memory to store operands, one could analyze to see, storing which operands would benefit the accelerator reduce network traffic. Storing all the operands would be the best, but it has an area overhead. Typically it is observed that accelerator on-chip memory consumes about 40-90% of accelerator area [22]. In an attempt to reduce the area, one can analyze the algorithm to determine which operands when stored can give maximum benefit for the invested area. Since accelerator implements a sequential design and works on multiprecision operands, the algorithm that accelerator implements greatly determines which operands when stored can give maximum benefit. The size of the internal memory could be increased incrementally to figure out an optimal memory size that would benefit each accelerator architecture.

APPENDIX A

HIGH LEVEL SPECIFICATION OF ACCELERATOR ARCHITECTURES

The high level specification of accelerator architectures in C, which are given as inputs to Vivado HLS tool are given below.

A.1 CIOS accelerator

```
#include "cios.h"

void cios(word x[N], word y, word out_r[N+1])
{ //Accumulates x[i]*y to out_r[0..N-1]
  //out_r[N] is generated (Not accumulated) as out_r[N] can be
   generated only in this iteration
  int i;
  static double_word SC = 0;
  static double_word XY = 0;
  static double_word sum1;
  static word x_read;
  static word out_read;
  static word SC_lower;
  static word C = 0;

  cios_loop: for(i=0; i<N; i++){

      x_read = x[i];
      out_read = out_r[i];

      XY = x_read * y;
      sum1 = out_read + XY ;
      SC = sum1 + C;

      C = (word)((SC & 0xffff0000) >> 16);
      SC_lower = (word)(SC & 0x0000ffff);
      out_r[i] = SC_lower;

  }
  out_r[N] = (word)((SC & 0xffff0000) >> 16);
}
```

Listing A.1: High level specification of CIOS accelerator

A.2 FIOS accelerator

```
#include "fios.h"

void fios(word x[N], word y, word m, word n[N], word out_r[N+1])
{
  int i;
  static double_word XY;
  static double_word MN;

  static double_word SC_xy;
  static word S_xy;
  static word C_xy;

  static double_word SC_mn;
  static word C_mn;

  static double_word SC;

  fios_loop: for(i=0; i<N; i++)
      {
          XY = x[i] * y;
          MN = m * n[i];

          SC_xy = out_r[i] + XY + C_xy;

          C_xy = (word)((SC_xy & 0xffff0000)>>16);
          S_xy = (word)(SC_xy & 0x0000ffff);

          SC_mn = S_xy + MN + C_mn;

          if (i==0)
            out_r[0] = S_xy;
          else
            out_r[i-1] = (word)(SC_mn & 0x0000ffff);
          C_mn = (word)((SC_mn & 0xffff0000)>>16);
      }

  SC = out_r[N] + C_mn + C_xy;
  out_r[N-1] = (word)(SC & 0x0000ffff);
  out_r[N] = (word)((SC & 0xffff0000)>>16);
}
```

Listing A.2: High level specification of FIOS accelerator

A.3 Proposed accelerator

```
#include "proposed_acc.h"

void proposed_acc(word x[N], word y[2], word out_r[N+3])
  { //Accumulates x[i]*y to out_r[0..N-1]
  //out_r[N] is generated (Not accumulated) as out_r[N] can be
    generated only in this iteration
  char i;
  static double_word S0C0_comb;
```

```
  static double_word S1C1_comb;
  static double_word S2C2_comb;

  static word S0;
  static word C0;
  static word S1;
  static word S2;
  static word C1;
  static word C2;

  static double_word AB0;
  static double_word AB1;

  static word AB0_low;
  static word AB0_high;
  static word AB1_low;
  static word AB1_high;

  static word y0;
  static word y1;

  static word out_read_0;
  static word out_read_1;
  static word out_read_2;

  static word y_read_0;
  static word y_read_1;

  static word x_read;

  short N_plus_1;
  short N_plus_2;

  C0 = 0;
  C1 = 0;
  C2 = 0;

  //Load S0,S1,S2
  out_read_0 = out_r[0];
  S0 = out_read_0;

  out_read_1 = out_r[1];
  S1 = out_read_1;

  y_read_0 = y[0];
  y_read_1 = y[1];

  y0 = y_read_0;
  y1 = y_read_1;

Inner_loop: for(i=0;i<N;i++)
  {
    out_read_2 = out_r[i+2];
    S2 = out_read_2;
```

```
62
63    x_read = x[i];
64
65    AB0 = x_read * y0;
66    AB1 = x_read *y1;
67
68    AB0_high = (word)((AB0 & 0xffff0000) >> 16);
69    AB0_low = (word)(AB0 & 0x0000ffff);
70
71    AB1_high = (word)((AB1 & 0xffff0000) >> 16);
72    AB1_low = (word)(AB1 & 0x0000ffff);
73
74    S2C2_comb = AB1_high + S2 + C2;
75    S1C1_comb = AB0_high + S1 + C1 + AB1_low;
76    S0C0_comb = AB0_low + S0 + C0;
77
78    S0 = (word)(S1C1_comb & 0x0000ffff);
79    S1 = (word)(S2C2_comb & 0x0000ffff);
80
81    C0 = (word)((S0C0_comb & 0xffff0000) >> 16);
82    C1 = (word)((S1C1_comb & 0xffff0000) >> 16);
83    C2 = (word)((S2C2_comb & 0xffff0000) >> 16);
84
85    out_r[i] = (word)(S0C0_comb & 0x0000ffff);
86  }
87
88      N_plus_1 = N + 1;
89      N_plus_2 = N + 2;
90
91      out_r[N] = S0 + C0;
92      out_r[N_plus_1] = S1 + C1;
93      out_r[N_plus_2] = C2;
94
95  }
```

Listing A.3: High level specification of proposed accelerator

APPENDIX B

ACCELERATOR IP IN ZYNQ SOC

The application code that the processor in Zync SoC runs, to communicate with the accelerator and and the rest of the SoC components, like BRAMs that store operands, is given below.

```
#include <stdio.h>
#include "platform.h"
#include "xcios_acc.h"
#include "xil_printf.h"
#include "xparameters.h"

int *XVecHW= (int *)0x40000000;
int *YVecHW = (int *)0x42000000;
int *OUTVecHW = (int *)0x44000000;

XCios_acc doCios_acc;
XCios_acc_Config *doCios_acc_cfg;

void init_cios_acc()
{
  int status = 0;
  doCios_acc_cfg = XCios_acc_LookupConfig(XPAR_CIOS_ACC_0_DEVICE_ID)
   ;
  if(doCios_acc_cfg)
  {
    status = XCios_acc_CfgInitialize(&doCios_acc,doCios_acc_cfg);
    if (status!=XST_SUCCESS)
      print("\n Failed to initialize \n");

  }
}
int main()
{
    init_cios_acc();
    printf("Test ACC\n\r");
  for (int i=0; i<32; i++ )
  {
    XVecHW[i] = 0x1789ffff + i;
  }
  for (int i=0; i<2 ; i++)
  {
```

```
36      YVecHW[i] = 0x23f80001 + i ;
37    }
38    for (int i=0; i< 35; i++)
39      {
40        OUTVecHW[i] = 0;
41      }
42      XCios_acc_Start(&doCios_acc);
43      while(!XCios_acc_IsDone(&doCios_acc));
44    for (int i=0; i<35;i++)
45    {
46      xil_printf("%x \t",OUTVecHW[i]);
47      if (i%8==0)
48        xil_printf("\n");
49    }
50    print("Done Test ACC\n\r");
51      return 0;
52  }
```

Listing B.1: Application program to run accelerator on Zynq SoC

BIBLIOGRAPHY

[1] IEEE standard specifications for public-key cryptography. *IEEE Std 1363-2000*, pages 1–228, 2000.

[2] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni. Securing hardware accelerators: A new challenge for high-level synthesis. *IEEE Embedded Systems Letters*, 10(3):77–80, Sep. 2018.

[3] Zhimin Chen and P. Schaumont. A parallel implementation of montgomery multiplication on multicore systems: Algorithm, analysis, and prototype. *IEEE Transactions on Computers*, 60(12):1692–1703, 2011.

[4] *Zedboard website*, accessed 13 November 2019. `http://zedboard.org/product/zedboard`.

[5] Dave Evans. The Internet of Things How the Next Evolution of the Internet The Internet of Things How the Next Evolution of the Internet Is Changing Everything. (April), 2011.

[6] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[7] *NIST Releases Draft Security Feature Recommendations for IoT Devices*, accessed 13 November 2019. https://www.nist.gov/news-events/news/2019/08/nist-releases-draft-security-feature-recommendations-iot-devices.

[8] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press (1996).

[9] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Cryptography in embedded systems: An overview. *Proc. Embedded World*, pages 735–744, 2003.

[10] *IDPrime 3811 Plug and Play Smart Card*, accessed 23 April 2019. `http://data-protection-updates.gemalto.com/files/2017/04/IDPrime3811_Plug_and_Play_Smart_Cards_PB_EN_v2_SEP082016_web.pdf`.

[11] *IDPrime PIV*, accessed 23 April 2019. `https://safenet.gemalto.com/multi-factor-authentication/idprime-piv-card/`.

[12] Thomas Kothmayr, Corinna Schmitt, Wen Hu, Michael Brünig, and Georg Carle. DTLS based security and two-way authentication for the Internet of Things. *Ad Hoc Networks*, 11(8):2710–2723, 2013.

[13] Kerry Maletsky. RSA vs ECC comparison for embedded systems. *White Paper, Atmel*, 5, 2015.

[14] Patrick Gallagher. Digital signature standard (DSS). *Federal Information Processing Standards Publications, volume FIPS 186-3*, 2013.

[15] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management part 1: General (revision 3). *NIST special publication*, 800(57):1–147, 2012.

[16] Gary L Miller. Riemann's hypothesis and tests for primality. In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 234–239. ACM, 1975.

[17] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation - Math. Comput.*, 44, 170:519–521, 1985.

[18] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[19] Jed Kao-Tung Chang, Chen Liu, and Jean-Luc Gaudiot. Hardware acceleration for cryptography algorithms by hotspot detection. In *International Conference on Grid and Pervasive Computing*, pages 472–481. Springer, 2013.

[20] *mbed TLS Library*, accessed 22 April 2019. `https://tls.mbed.org/`.

[21] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014.

[22] Y Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, 2015.

[23] Thomas Blum and Christof Paar. Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*, pages 70–77. IEEE, 1999.

[24] Nathaniel Ross Pinckney and David Money Harris. Parallelized radix-4 scalable montgomery multipliers. In *SBCCI*, 2007.

[25] Colin D. Walter. Systolic modular multiplication. *IEEE transactions on computers*, 42(3):376–378, 1993.

[26] Florent Bernard. Scalable hardware implementing high-radix montgomery multiplication algorithm. *Journal of Systems Architecture*, 53(2-3):117–126, 2007.

[27] C. McIvor, M. McLoone, and J. V. McCanny. High-radix systolic modular multiplication on reconfigurable hardware. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pages 13–18, Dec 2005.

[28] YS Kim, WS Kang, and JR Choi. Implementation of 1024-bit modular processor for rsa cryptosystem, School of Electronic and Electrical Engineering, Kyungpook National University, 1370 Sankyok-Dong, Book-Gu, Taegu, Korea. pages 702–701.

[29] Viktor Bunimov, Manfred Schimmler, and Boris Tolg. A complexity-effective version of montgomerys algorithm. In *in Workshop on Complexity Effective Designs, ISCA02, May 2002, http://www.ee.rochester.edu:8080/ albonesi/wced02*, 2002.

[30] C. McIvor, M. McLoone, and J. V. McCanny. Fpga montgomery multiplier architectures - a comparison. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 279–282, April 2004.

[31] Adrien Le Masle, Wayne Luk, Jared Eldredge, and Kris Carver. Parametric encryption hardware design. In *International Symposium on Applied Reconfigurable Computing*, pages 68–79. Springer, 2010.

[32] S. Kuang, J. Wang, K. Chang, and H. Hsu. Energy-efficient high-throughput montgomery modular multipliers for rsa cryptosystems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(11):1999–2009, Nov 2013.

[33] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 119–132, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[34] Erich Wenger. A lightweight atmega-based application-specific instruction-set processor for elliptic curve cryptography. In Gildas Avoine and Orhun Kara, editors, *Lightweight Cryptography for Security and Privacy*, pages 1–15, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[35] Erich Wenger, Thomas Unterluggauer, and Mario Werner. 8/16/32 shades of elliptic curve cryptography on embedded processors. In *International Conference on Cryptology in India*, pages 244–261. Springer, 2013.