สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

HARDWARE DESIGN OF HYBRID (RSA-IDEA) CRYPTOSYSTEM

MEDI A. NAZAR

เลขหมู่..............................
เลขทะเบียน 44512
วัน,เดือน,ปี - 9 ก.พ. 2549

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
MASTER OF ENGINEERING IN COMPUTER ENGINEERING
SCHOOL OF GRADUATE STUDIES
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2004

ISBN 974-15-1340-2

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

.b..............................
.i..............................

COPYRIGHT 2004
SCHOOL OF GRADUATE STUDIES
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

Forbidden to modify the content, and cite the document when use.

หัวหน้าวิทยานิพนธ์	**การออกแบบฮาร์ดแวร์สำหรับการเข้ารหัสแบบผสมระหว่างอาร์เอสเอและไอเดีย**
นักศึกษา	**นางสาวเมดี นาซาร์**
รหัสประจำตัว	45067101
ปริญญา	**วิศวกรรมศาสตรมหาบัณฑิต**
สาขาวิชา	**วิศวกรรมคอมพิวเตอร์**
พ.ศ.	2547
อาจารย์ผู้ควบคุมวิทยานิพนธ์	**รศ. ประทีป บัญญัติินพรัตน์**
อาจารย์ผู้ควบคุมวิทยานิพนธ์ร่วม	**ดร. สุรินทร์ กิตติธรกุล**

บทคัดย่อ

อัลกอริธึมสำหรับการเข้ารหัสแบบสมมาตร เช่น อัลกอริธึมไอเดีย (IDEA) และอัลกอริธึมสำหรับการเข้ารหัสแบบไม่สมมาตร ยกตัวอย่างเช่น อัลกอริธึมอาร์เอสเอ (RSA) อาศัยการคูณแบบมอดูลาร์ของกุญแจและข้อมูลเป็นจำนวนมาก โดยความปลอดภัยของข้อมูลนั้นจะเพิ่มตามขนาดของกุญแจ การใช้ฮาร์ดแวร์เฉพาะสามารถช่วยให้การคำนวณเร็วขึ้นเมื่อต้องการเข้ารหัสด้วยความเร็วสูง แต่เมื่อขนาดของกุญแจใหญ่ขึ้นความถี่สัญญาณนาฬิกาของฮาร์ดแวร์เฉพาะอาจลดลงถ้าออกแบบไว้ไม่ดีพอ วิทยานิพนธ์ฉบับนี้นำเสนออะเรย์ของตัวประมวลผลแบบซิสตอลิคสมบูรณ์ (Fully Systolic) สำหรับการคูณแบบมอดูลาร์บนอุปกรณ์ชนิดเอฟพีจีเอซึ่งการคูณแบบมอดูลาร์ที่ใช้เป็นวิธีการของมอนโกเมอรี ผลงานของเราแสดงให้เห็นว่า ความถี่สัญญาณนาฬิกามีค่าคงที่ถึงแม้ว่าขนาดของกุญแจจะเพิ่มขึ้นก็ตาม สถาปัตยกรรมของเราทำงานที่ความถี่สัญญาณนาฬิกาสูงกว่าผลงานอื่นๆ ที่ไม่เป็นแบบซิสตอลิคอย่างสมบูรณ์ ขณะเดียวกัน ผลคูณของพื้นที่และเวลาก็ต่ำกว่า สุดท้ายนี้ เวลาในการคำนวณทั้งหมดและเวลาก่อนจะได้ผลลัพธ์บิทแรกของการคำนวณผลคูณแบบมอดูลาร์ของกุญแจขนาด l บิท คือ $3l+1$ และ $2l+1$ ลูกคลื่นนาฬิกาตามลำดับ

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

Thesis	Hardware Design of Hybrid (RSA-IDEA) Cryptosystem
Student	Miss Medi A. Nazar
Student ID	45067101
Degree	Master of Engineering
Programme	Computer Engineering
Year	2004
Thesis Advisor	Assoc. Prof. Pratheep Bunyatnoparat
Co-Advisor	Dr. Surin Kittitornkun

ABSTRACT

Public-key cryptographic algorithms such as RSA algorithm require modular multiplications of very large operands. In RSA, the higher security the larger operand size which may reduce the clock rate and result into lower encryption throughput. This paper presents a fully systolic linear-array for the computation of Montgomery modular multiplication that is implemented using FPGA. Our fully systolic design shows that a high and nearly constant clock rate is achievable regardless of the size of the operand. As compared with the non-fully systolic architecture, our design offers a higher operating frequency that yields a higher throughput rate and a lower area-time product. The total execution time for an l-bit modular multiplication is $3l+1$ cycles, latency of $2l+1$ cycles and a throughput of one bit per clock cycle, where l is the length of the modulus.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Acknowledgements

This will take a little longer than ordinary acknowledgment you used to read, please bear with me while reading it because I want to use this opportunity to thank everybody whom I believe have helped me and taught me in one way or another throughout the course of my master's study.

First of all I would like to thank ASEAN University Network/Southeast Asia Engineering Education Development Network (AUN/SEED-Net) for the financial support of my master's study here in Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang (KMITL), Thailand. I would also like to extend my appreciation to Dr. Fujiwara, Mr. Iwadate, Ms. Meena and Ms. Siriporn of AUN/SEED-Net for their effort and understanding when I asked for an extension.

I would like to thank Dr. Julius Maridable, dean of the College of Engineering, De La Salle University (DLSU), Manila, Philippines for encouraging me to take this scholarship opportunity and for the continuous support of my colleagues from the Department of Electronics and Communications Engineering, namely; Mr. Edwin Sybingco, Mr. Roderick Yap, Dr. Felixberto Cruz, Mr. Oswald Sapang, Mrs. Antonette, Sir Neil, Mosses, Edsel, Therese, Angelo, Tristan and to the rest of the faculty, thank you very much, I'm very excited to go back and work again with you guys.

My deepest thanks go to my co-advisor Dr. Surin Kittitornkun, who had been very active in giving me guidance, patience, encouragement, and motivation. His advices and reminders drove me to the finish line of this race. I admire him as a professor and an advisor because of his enthusiastic and zealous nature towards learning, which I believe to be an important character that people in the academe should have. Thank you very much for your time and ideas especially when I really feel discouraged and when my brain doesn't feel like working.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

I thank Dr. Pratheep Bunyatnoparat, my advisor, for his active participation in our thesis fund proposal and the budgeting, and Dr. Jongkol, AUN/SEED-Net coordinator here in KMITL.

To my professors in my coursework; Ajan Surin, Ajan Aranya & Somsak Walàiracht, Ajan Mitatha, Ajan Boontee, Ajan Pratheep, Ajan Ouen, Ajan Kritawan and Ajan Chutimet, thank you for sharing to me your knowledge and giving me inspiration with regards to teaching.

Dr. Anchaleeporn, who fetched me from the airport and took care of me during my first few weeks here in KMITL, Po and Pu, her nieces, who are very friendly and kind as well as their group of friends, thank you very much for all the assistance and favors you've done for me.

I would like to thank Tukta and her officemates from the Graduate Studies of the Faculty of Engineering, Pi Lek and her staff from the Graduate Studies Office, Pi Pu, Dr. Jongkol's secretary, and Pi Eh, one of the staff in Computer Engineering office who's very friendly and approachable whenever I go there.

To my Lao co-scholars, Ajan Nouanchanch who had been my interpreter and housemate during my first year here in KMITL and for inviting me to her home in Laos, Khampheth who's always there ready to lend me his ears when I want to say something, Ajan Tha, Kham, Ting and Ajan Somphon, thank you for your friendship and kindness.

To Chau, my Vietnamese co-scholar, and Ovie, my Indonesian friend from ABAC, thank you for your support during my defense, it means a lot to me.

To the master's students of Computer Engineering especially those who stay in room 804, thank you very much for the assistance and accompanying me whenever I need a translator, you've been part of my daily life for two years, I enjoyed the times we went out together. I want to apologize if I behaved differently and though there are dull times on the latter part of my stay there, I still want to thank you for that because I learned something

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

from you guys. To Cat, who had always been concern with my situation, very supportive and helpful whenever I need assistance, thank you very much.

To the keeper of 8th floor of ECC building and the lady in photocopy shop in the ground floor, thanks for greeting me with your smiles whenever we see each other. To the staff of Kaeda Cafeteria and Unna Coffee Shop, my hideout whenever I feel hungry, thank you for being kind to me. To the staff of WWW Bookstore whose been friendly to me and used to give me discounts, thank you so much.

To Ajan Jing and Beth, my only fellow Filipino here in KMITL and their friend Mommy Pean, who also became my friend, thank you thank you very much for the trip to Chaiang Mai, I really had a great time. I won't forget the first time I slept in tent on top of the mountain Doi Inthanon, my brain had been frozen that night!

And to my Filipino fellow whom I met here in Thailand especially the members of Singles for Christ, thank you very much for the friendship, companionship, for giving me the chance to participate in our activities, I had a lot of fun and memories to bring home. Thanks for helping me to grow spiritually and for being part and touching my life.

To my dearest Filipino neighbor here in Sansabuy Complex; Shella, Ralph and Fe, my extended family where I always eat and watch TV. Thank you for lending me your ears whenever I need to release my anger, disappointments and share the daily happening in my life within one year.

To my college *barkada* (a circle of very close friends) Cel, Tere, She, Allan, Jake, Carl, Joy and Jem, thank you for the constant communication we have, for being always there whenever I need somebody to talk to, I'm really glad I met you and became my real friends. To my former college classmates and batchmates, thank you for having once in awhile conversation over the Net. To my high school *barkada*, Fer, Rhea, Juliana, Ian, Raquel and Weng, thank you for the prayers and updates, I'm excited to see you guys. To my friends since elementary: Seh, Tess, Divina and Jonathan, thank you for always

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

believing in me, for the non-changing environment whenever I visit there in our small town, and for being part of my life throughout these years. To my other email, friendster and YM buddies, Jojo, Renz, Mon, Tina, Melvin, Ate Rhea, Ate Mepa, Are Ariane, Ate Monet, and Cathy, thanks for the time you spent talking to me, you just don't know how you made my life normal in the midst of pressure and disappointments I've gone through.

And the last but not the least, to my family who had been one source of my inspiration, thank you for always giving me moral support and for always praying for me, who always show their concern and love to me even if I was far from them.

All honor and praise, I want to lift it up to Almighty God, Who has been my source of strength and wisdom. Thank You for sending all the above mentioned people who have helped me in this chapter of my life. And most of all, thank You for loving me and being with me always.

Medi A. Nazar

King Mongkut's Institute of Technology Ladkrabang

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table of Contents

Page

Thai Abstract I

English Abstract II

Acknowledgements...III

Table of Contents VII

List of Tables X

List of Figures XII

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Introduction to Cryptography 3

1.2.1 Types of Cryptography 3

1.2.2 Cryptographic Algorithms 4

1.3 Thesis Organization 5

Chapter 2 Cryptographic Algorithms and FPGA 6

2.1 International Data Encryption Algorithm (IDEA) 6

2.1.1 Primitive Operations 6

2.1.2 Key Expansion 8

2.1.3 IDEA Algorithm...8

2.2 RSA ALGORITHM 10

2.2.1 Key Generation 11

2.2.2 RSA Operations 11

2.2.3 Modular Exponentiation Operation 12

2.2.4 Modular Multiplication Operation 14

This material is reserved for educational purpose only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table of Contents (continued)

Page

2.2.4.1 Interleaving Multiplication and Reduction 15

2.2.4.2 Montgomery's Algorithm 16

2.3 Field Programmable Gate Arrays (FPGAs) 18

2.3.1 FPGA Technologies and Their Suitability for Modular Arithmetic 19

2.3.1.1 Xilinx FPGA Architectural Analysis 20

2.3.1.2 FPGA Architecture Consideration for Public-Key Algorithm Implementation 21

Chapter 3 Previous Works on Modular Multiplication 23

3.1 Modular Arithmetic 23

3.2 Modular Multiplication 25

3.2.1 Systolic Array Architectures [2, 6] 25

3.2.2 Different Architectures and FPGA Implementations 26

3.2.2.1 Sequential vs. Parallel Hardware Implementations [3]...............26

3.2.2.2 Resource vs. Speed Efficient Architecture [1] 30

3.2.2.3 FPGA Architecture-Dependent Design [8] 35

Chapter 4 Systolic Design Methodology 40

4.1 Montgomery Modular Multiplication Algorithm 40

4.2 Mapping Procedure [13] 43

4.2.1 Algebraic Mapping Procedure 43

4.2.2 Mapping DG into SFG 44

4.2.3 Systolic Mapping 46

4.3 VHDL Design and FPGA Implementation 48

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table of Contents (continued)

Page

4.3.1 Architectural Design and Simulation 48

4.3.2 FPGA Implementation Results 52

Chapter 5 Conclusion and Future Works 58

5.1 Conclusion 58

5.2 Future Works 59

References 60

Appendix A VHDL Design of the Systolized SFG-II 64

Appendix B Testbench and Simulation results 85

Appendix C Synthesis Results 87

Appendix D An FPGA Implementation of Systolic Array for Montgomery Modular Multiplication 92

Author's Biography....................99

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

List of Tables

Table | Page

3.1 Computation of R + a_iB +q_iM (MB = M+B, precomputed)[3]............. 27

3.2 Iterative vs. systolic Montgomery modular multiplier [3]..................... 29

3.3 CLB usage, minimal clock cycle time, and time-area product of modular exponentiation architectures on Xilinx FPGAs[1] 32

3.4 CLB usage and execution time for a full modular exponentiation[1] 32

3.5 Application to RSA: Encryption [1] 32

3.6 CLB usage(Area), minimal clock cycle time(T), and time-area(TA) product of modular exponentiation architectures on Xilinx FPGAs [1] ... 33

3.7 CLB usage and execution time for a full modular exponentiation of design 1(where u=4) and design 2 [1]... 34

3.8 Application to RSA: Encryption[1] 34

3.9 Speed & area results for the pipelined Mongtomery modular multiplier[8] 38

3.10 Speed & area results for a non-pipelined Mongtomery modular multiplier[8] 38

3.11 Speed & area results for the pipelined RSA encryptor/decryptor [8] 39

4.1 Computation of P + a_iB +q_iN (NB = N + B, precomputed) 41

4.2 Design Parameter, our design vs. Nedjah's design [3]......................... 51

4.3 Performance figures, our design vs. Nedjah's design [3]..................... 52

4.4 Design Parameter Comparison..54

4.5 Performance Comparison: Speed 54

4.6 Performance Comparison: Area 55

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

List of Tables (continued)

Table Page

4.7 Performance Comparison: Area-Time Product 56

4.8 Performance Comparison:

Time for One Modular Multiplication in μs 57

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

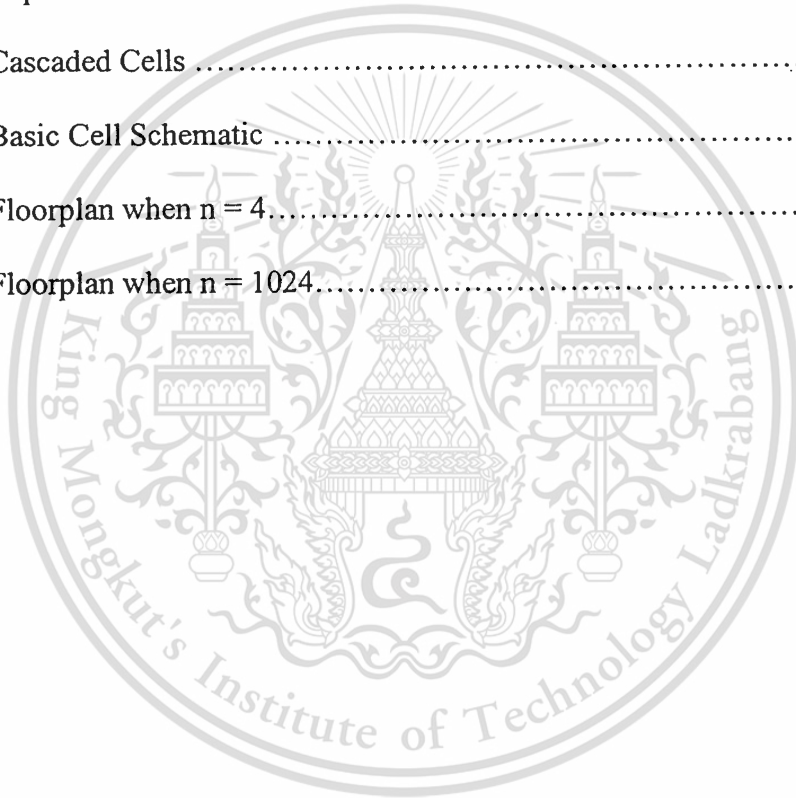
List of Figures

Figure		Page
2.1	Block diagram of the IDEA Algorithm	7
2.2	IDEA ciphering computational graph	10
3.1	Systolic architecture of Montgomery multiplier [3]	28
3.2	Right border PEs of mxm array[3]	29
3.3	Top-most PEs of mxm array [3]	29
3.4	Processing Element (unit) of resource efficient architecture [1]	31
3.5	Processing Element (unit) of speed efficient architecture [1]	34
3.6	Montgomery modular multiplier architecture with single adder and multiplexer [8]	35
3.7	Pipeline modular multiplier [8]	36
3.8	Pipelined Multiplication: DDG using the pipelined multiplier units [8]	37
4.1	Dataflow within the Dependence Graph based on Algorithm 4.3	42
4.2	a) Simple Dependence Graph(DG) represention of Systolic Montgomery Algorithm, b) SFG-I, with a projection vector [0 1], c) SFG-II, with a projection vector [1 0]	45
4.3	a) Systolized SFG-I, b) Systolized SFG-II	48
4.4	a) Architecture designof Systolized SFG-II when n=4	50
4.4	b)Typical structure of each processing element in systolized SFG-II	51
4.5	Clock cycle time: Nedjah's multiplier vs. Systolized SFG-II multiplier	53
4.6	Area-Time product: Nedjah's multiplier vs. Systolized SFG-II multiplier	53
4.7	Performance Comparison: Speed	55

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

List of Figures (continued)

Figure		Page
4.8	Performance Comparison: Area	56
4.9	Performance Comparison: Area-Time Product	56
B.1	a) Output waveform of first simulation (Testbench1)	
	b) Output waveform of second simulation (Testbench2)	86
C.1	Top level Schematic	87
C.2	Cascaded Cells	88
C.3	Basic Cell Schematic	89
C.4	Floorplan when n = 4	90
C.5	Floorplan when n = 1024	91



This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Chapter 1

Introduction

1.1 Motivation

The popularity of the Internet and portable, battery-operated computing terminals and devices drive towards a global network that allows users to communicate and share information with other systems located around the globe. Hence, wireless networks have become prevalent nowadays and will continue to grow according to market research. The use of e-commerce and electronic banking that is becoming popular over the Internet, a trend that will invariably migrate to wireless networks, shows that insecure networks lead not only to fraud and invasion of privacy but to monetary risks as well, since wireless networks use air as the transmission medium which is susceptible to tampering and eavesdropping. Hence, security issues arise that needs a lot of attention.

Secured cryptographic implementations become one of the strong requirements for commercial Internet. There have been many software-based cryptographic products developed to assure transactions and other vital information on the Internet. Today, hardware-based cryptographic implementations have become one of the interests in current research in order to strengthen security and improve performance.

Software-based solution is energy/computationally inefficient for certain cryptographic algorithms, particularly the asymmetric algorithms which require long computation. Furthermore, with the migration to portable battery-operated mobile computing terminals, a software-based implementation needs to be re-evaluated due to both

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

the energy and processing power constraints in a portable battery-operated environment. Another problem with software-based solutions is that software running in an open environment is untrustworthy as both the code and secrets used to implement cryptographic algorithms must be stored in memory external to the processor, making it susceptible to a variety of security attacks.

Hardware-based cryptographic implementation solution can remedy the aforementioned weaknesses of software-based solution for performing operations. With hardware solution, attacks become much more difficult as the secrets can be contained within the processor using nonvolatile memory that is externally inaccessible. It can also be done in dedicated hardware implementations thereby making them very attractive for energy-constrained applications such as the computing portable terminals and other mobile devices. The use of a dedicated cryptographic hardware coprocessor also offloads the heavy computational demands of cryptographic algorithms from the embedded general purpose processor, freeing it to perform other tasks to which it is better suited.

For hardware implementations, reconfigurable technology such as the Field Programmable Gate Array (FPGA) offers many advantages over semi-custom Application Specific Integrated Circuits (ASICs). FPGAs assure a short time to the market, high flexibility including capability for frequent modifications of hardware, low development cost and low cost of the final product. It has the potential for fast, low cost reprogramming and experimental testing of a large number of various architectures and revised versions of the same architecture. One of the goals of this research is to develop a hardware design for cryptographic algorithm that will be implemented on FPGA.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

1.2 Introduction to Cryptography

Cryptography is the science of encoding messages in such a way that unauthorized parties cannot decipher the encoded information in a reasonable amount of time. In the past, the field of cryptography was primarily the regime of the military, who used it for providing secure communication channels in hostile environments. Today, cryptography has become more of a public science due to its increased use in digital communications to provide security. Formal methods have been developed and refined for both the construction and analysis of cryptographic algorithms which include mathematical foundations. This section attempts to provide a brief introduction to the field of cryptography.

1.2.1 Types of Cryptography

There are two basic types of cryptographic algorithms; asymmetric and symmetric. Asymmetric cryptographic algorithms do not require any secret information to be shared between the communicating parties. They rely on the existence of mathematical functions that have the property that they can be computed efficiently (i.e., in polynomial time), but are computationally infeasible to invert without knowing some secret piece of information. The asymmetry is exploited to form cryptographic algorithms which utilize two keys: *public key*, used for encoding the data (encryption) and the *private key*, used for recovering the data that has been encoded (decryption). Public keys are openly stored so that anyone can encrypt a message. Asymmetric algorithms are commonly referred to as *public key algorithms*.

Because of the number-theoretic properties of the algorithms used, only the intended recipient who generated the public-private key pair can decode the message correctly. The underlying mathematics which enables this asymmetry requires a great deal more computation, which limits the degree of optimizations that can be applied to improve performance, thus not a good choice for encrypting large amounts of data.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Symmetric algorithms on the other hand derive their security from a secret piece of information that is shared by the communicating parties, commonly referred to as the *secret key*, that is why symmetric algorithms are typically referred to as *secret key cryptography*. The primary benefit of using secret key cryptography is that the shared secret can be exploited to create algorithms that operate very efficiently in terms of their computational complexity. Thus, public-key algorithms are used primarily for establishing secret keys throughout the network in a secure manner, as well as for user authentication and identification while symmetric algorithms are used to encrypt the bulk of data.

There are two types of secret key algorithms, these are *block* and *stream* ciphers. Block ciphers are symmetric key algorithms that operate on blocks of data, n bits at a time, to generate an m-bit output that forms the encrypted message. Typically, $m = n$ to avoid any data expansion. As a result, a block cipher can be thought of as a memoryless n-bit permutation of the inputs under the influence of the secret key. Stream ciphers on the other hand contain internal state that makes their output a time-dependent function, thereby avoiding the replay weakness that haunts block ciphers. In addition, a stream cipher operates on a data stream, typically a single bit wide, rather than a block of data.

1.2.2 Cryptographic Algorithms

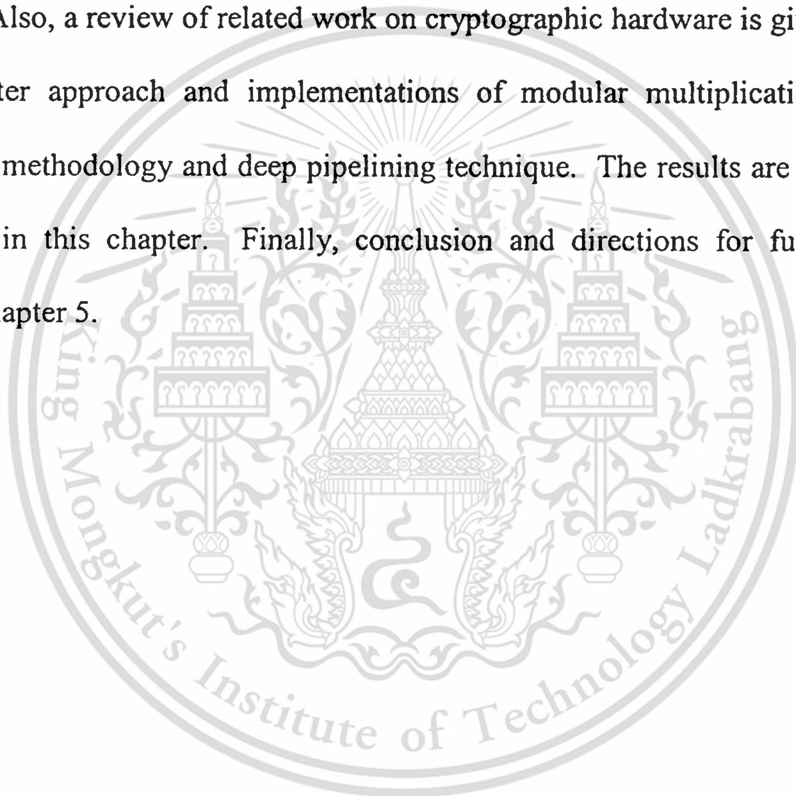
International Data Encryption Algorithm (IDEA) is an example of symmetric algorithm which considered being the most secure cipher due to its immunity to attacks [9]. On the other hand, Rivest-Shamir-Adleman (RSA) algorithm is an asymmetric or public key algorithm which is the best known, most versatile, and widely used public key cryptosystem today. The combination of RSA and IDEA has been used in freeware email encryption system PGP (Pretty Good Privacy). As previously discussed, symmetric algorithms are commonly used for encrypting the bulk of data, while public-key algorithms

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

are used for digital signature and encryption/decryption of secret keys used in symmetric algorithms using the private-key/public keys.

1.3 Thesis Organization

In Chapter 2, cryptographic algorithms such as IDEA and RSA algorithms are discussed in detail, their basic operation and the key generation. An overview of modular arithmetic has been discussed which is the most important operation used in IDEA and RSA is presented in Chapter 3. Also, a review of related work on cryptographic hardware is given. Chapter 4 presents a better approach and implementations of modular multiplication which use systolic design methodology and deep pipelining technique. The results are also presented and evaluated in this chapter. Finally, conclusion and directions for future work are presented in Chapter 5.



This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Chapter 2

Cryptographic Algorithms and FPGA

In this chapter, IDEA and RSA cryptographic algorithms will be introduced. It will be shown that the primary operation used for both algorithms is modular multiplication. Lastly, the Field Programmable Gate Array (FPGA) analysis and consideration for modular arithmetic will be discussed.

2.1 International Data Encryption Algorithm (IDEA)

International Data Encryption Algorithm (IDEA) was originally called IPES (Improved Proposed Encryption Standard). It was developed by Xuejia Lai and James L. Massey of ETH Zuria [10]. It is a symmetric block cipher using a 128-bit key.

IDEA is based on some impressive theoretical foundations and, although cryptanalysis has made some progress against reduced round variants, the algorithm still seems strong. Bruce Schneier [9], believes that IDEA is the best and most secure block algorithm available to the public at the time he wrote his book.

2.1.1 Primitive Operations

IDEA encrypts a 64-bit block of plaintext into 64-bit block of ciphertext using a 128-bit key. The plaintext block is divided into four 16-bit sub-blocks X_1, X_2, X_3, X_4 then undergoes a series of transformation to scramble the text. The same algorithm is used for both encryption and decryption.

for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

As with all the other block ciphers, IDEA uses both confusion and diffusion. The design philosophy behind the algorithm is one of "mixing operations from different algebraic groups." Three algebraic groups are being mixed, and they are all easily implemented in both hardware and software [9]:

(1) XOR,

(2) Multiplication modulo $(2^{16}+1)$,

(3) Addition modulo 2^{16}.

Each primitive operation in IDEA maps two 16-bit quantities (plaintext sub-blocks and subkey) to a 16-bit quantity (ciphertext). Basically it consist of 8 rounds followed by an output transformation, see Figure 2.1.

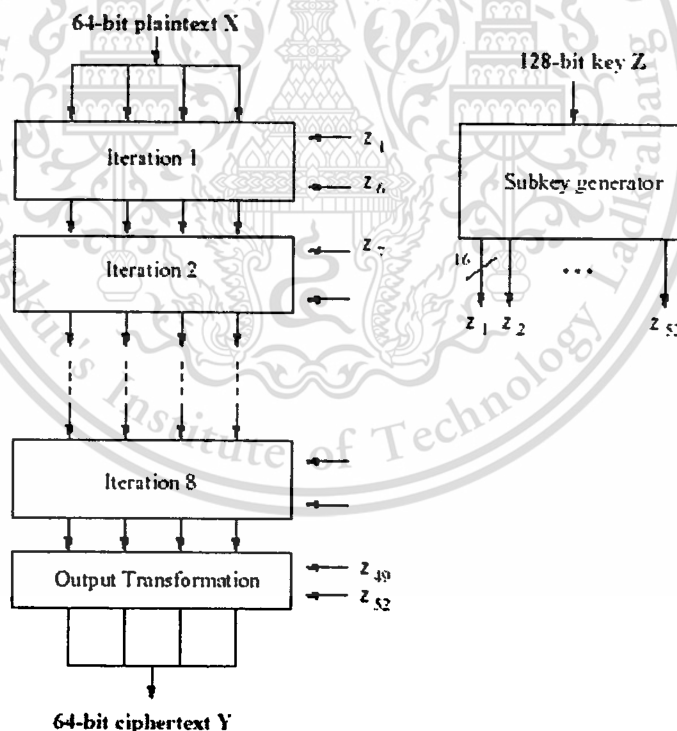


Figure 2.1 Block diagram of the IDEA Algorithm.

Multiplication in IDEA is done by first calculating the 32-bit result, and then taking the remainder when divided by $(2^{16}+1)$. Multiplication mod $(2^{16}+1)$ is reversible, in the

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

sense that every number x between 1 and 2^{16} has an inverse y (i.e., a number in the range 1 to 2^{16} such that multiplication by y will undo multiplication by x), because $(2^{16}+1)$ happens to be prime. This is one subtlety, though. The number 0 which can be expressed in 16 bits, haven't have an inverse, and the number 2^{16}, which is in the proper range of mod $(2^{16}+1)$ arithmetic, cannot be expressed in 16 bits. So both problems are solved by treating 0 as an encoding for 2^{16}.

2.1.2 Key Expansion

The 128-bit key is expanded into 52 16-bit keys, Z_1, Z_2, ... Z_{52}. These are generated from the 128-bit subkey as follows:

- The 128-bit key is split into eight 16-bit subkeys. These are the first eight subkeys for the algorithm, six for the first round and the first two for the second round, see Figure 2.1.
- Then the 128-bit key are shifted 25 bits to the left to make a new subkeys which is split into the next eight 16-bit subkeys. The first four are used in round 2; the last four are used in round 3.
- The 128-bit key is again shifted 25 bits to the left to make the next eight subkeys, and so on until the end of the algorithm.

2.1.3 IDEA Algorithm

The four plaintext sub-blocks become the input to the first round of the algorithm. In each round the four sub-blocks are XORed, added, and multiplied with one another with the six 16-bit subkeys. Between rounds, the second and the third sub-blocks are swapped. Finally, the four sub-blocks are combined with four sub-keys in an output transformation, see Figure 2.2. In each round, the sequence of events is as follows:

(1) Multiply X1 and the first subkey.

This material is used for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

(2) Add X2 and the second subkey.

(3) Add X3 and the third subkey.

(4) Multiply X4 and the fourth subkey.

(5) XOR the results of steps (1) and (3)

(6) XOR the results of steps (2) and (4).

(7) Multiply the results of step (5) with the fifth subkey.

(8) Add the results of steps (6) and (7).

(9) Multiply the results of step (8) with the sixth subkey.

(10) Add the results of steps (7) and (9).

(11) XOR the results of steps (1) and (9).

(12) XOR the results of steps (3) and (9).

(13) XOR the results of steps (2) and (10).

(14) XOR the results of steps (4) and (10).

The notation $Z_i^{(r)}$ in Figure 2.2 are the subkeys, where *i* is the number of subkey used in each round and *r* is the number of round. X_1, X_2, X_3, X_4 are the plaintext sub-blocks while Y_1, Y_2, Y_3, Y_4 are the ciphertext blocks.

The output of the round is the four sub-blocks that are the results of steps (11), (12), (13), and (14). Swapping the two inner blocks (except for the last round), and that's the input of the next round. After the eighth round, there is a final output transformation:

(1) Multiply X0 and the first sub-key.

(2) Add X1 and the second sub-key.

(3) Add X2 and the third sub-key.

(4) Multiply X3 and the fourth sub-key.

Finally, the four sub-blocks are reattached to produce the ciphertext.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

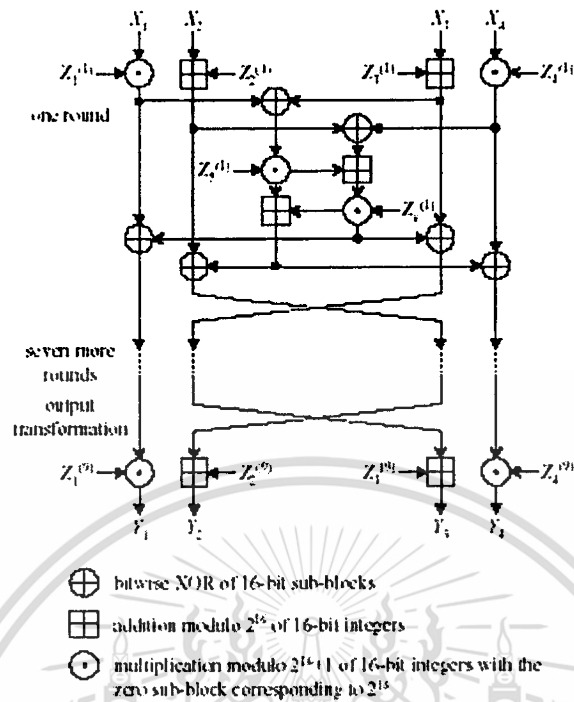


Figure 2.2 IDEA ciphering computational graph.

Decryption is exactly the same, except that the subkeys are reversed and slightly different. The decryption subkeys are either the additive or multiplicative inverses of the encryption subkeys.

2.2 RSA ALGORITHM

The RSA Algorithm was developed by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978. RSA scheme is asymmetric block cipher in which the plaintext and ciphertext are integers between 0 and n-1 for some n. A typical size for n is 1024 bits, or 309 decimal digits. The RSA algorithm can be used for both public cryptosystem and digital signatures. Its security is based on the difficulty of factoring large integers.

For digital signature, the private key is used for signing and the public key is used for verification while in public cryptosystem, public key is used for encryption and the private key is used for decryption.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

2.2.1 Key Generation

The parameters in RSA are N, p and q, e, and d [11]. The modulus N is the product of the distinct large random primes p and q: $N = pq$.

The public exponent e is a number in the range $1 < e < \varphi(N)$ such that $\gcd(e, \varphi(N)) = 1$,

where $\varphi(N)$ is Euler's totient function of N, given by, $\varphi(N) = (p-1)(q-1)$.

The private exponent d is obtained by inverting e modulo $\varphi(N)$; $d = e^{-1} \bmod \varphi(N)$, can be rewritten as, $d*e = 1 \bmod \varphi(N)$ using the extended Euclidean algorithm. Usually one selects a small public exponent, e.g., $e = 2^{16} + 1$.

Modulus (N) and encryption key (e) are publicly published while decryption key (d) and random primes p and q are kept secret.

2.2.2 RSA Operations

The RSA algorithm requires computation of the modular exponentiation which is broken into a series of modular multiplications by the application of exponentiation heuristics. Before getting into the details of these operations, here are the following definitions:

- The public modulus N is a k-bit positive integer, ranging from 512 to 2048 bits.
- The secret primes p and q are approximately k=2 bits.
- The public exponent e is an h-bit positive integer. The size of e is small, usually not more than 32 bits. The smallest possible value of e is 3.
- The secret exponent d is a large number; it may be as large as $\varphi(N)$-1. We will assume that d is a k-bit positive integer.

Once the modulus (N), the public (e) and private (d) exponents are determined, the senders and recipients perform a single operation for signing, verification, encryption, and decryption. The operation required is the modular exponentiation.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

The encryption operation is performed by computing;

$$C = M^e \pmod{N}$$

where M is the plaintext such that $0 \le M < n$. The number C is the ciphertext from which the plaintext M can be computed using;

$$M = C^d \pmod{N}.$$

Hence, e and d had been called as public and private exponents.

In public key cryptography algorithms, the essential arithmetic operation is modular multiplication, which is used to calculate modular exponentiation. However, modular exponentiation on numbers of hundreds of bits (512 bits or higher) makes it difficult for the RSA algorithm to attain high throughput.

2.2.3 Modular Exponentiation Operation

The modular exponentiation operation is simply an exponentiation operation where multiplication and squaring operations are modular operations. The exponentiation heuristics developed for computing M^e are applicable for computing $M^e \pmod{N}$. The binary method for computing $M^e \pmod{N}$ given the integers M, e, and N has two variations depending on the direction by which the bits of e are scanned: Left-to-Right (LR) and Right-to-Left (RL). The LR binary method is more widely known:

```
Input: M; e; N
Output: C := M^e mod N
1.      if e_{h-1} = 1
                then C := M
        else C := 1
2.      for i = h - 2 downto 0
2a.             C := C * C (mod N)              \\square
2b.             if e_i = 1
                        then C := C * M (mod N)   \\ multiply
3.      return C
```

The bits of e are scanned from the most significant to the least significant, and a modular squaring is performed for each bit. A modular multiplication operation is

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

performed only if the bit is 1. An example of LR binary method is illustrated below for h = 6 and e = 55 =(110111). Since $e_5 = 1$, the LR algorithm starts with $C := M$, and proceeds as;

i	e_i	Step2a (C)	Step2b (C)
4	1	$(M)^2 = M^2$	$M^2 * M = M^3$
3	0	$(M^3)^2 = M^6$	M^6
2	1	$(M^6)^2 = M^{12}$	$M^{12} * M = M^{13}$
1	1	$(M^{13})^2 = M^{26}$	$M^{26} * M = M^{27}$
0	1	$(M^{27})^2 = M^{54}$	$M^{54} * M = M^{55}$

The RL binary algorithm, on the other hand, scans the bits of e from the least significant to the most significant, and uses an auxiliary variable P to keep the powers M.

```
Input: M; e; N
Output: C := M^e mod N
1.      C := 1 ; P := M
2.      for i = 0 to h-2
2a.             if e_i = 1
                        then C := C*P (mod N)
2b.             P := P*P (mod N)
3.              if e_{h-1} = 1
                        then C := C*P (mod N)
4.      return C
```

The RL algorithm starts with C =: 1 and $P := M$, proceeds to compute M^{55} as follows:

i	e_i	Step2a (C)	Step2b (P)
0	1	$1 * M = M$	$(M)^2 = M^2$
1	1	$M * M^2 = M^3$	$(M^2)^2 = M^4$
2	1	$M^3 * M^4 = M^7$	$(M^4)^2 = M^8$
3	0	M^7	$(M^8)^2 = M^{16}$
4	1	$M^7 * M^{16} = M^{23}$	$(M^{16})^2 = M^{32}$
Step 3: $e_5 = 1$, thus $C := M^{23} * M^{32} = M^{55}$			

Comparison between LR and RL algorithm in terms of time and space requirements:

- Both methods require h-1 squarings and an average of 1/2 (h-1) multiplications.
- The LR binary method requires two registers: M and C.
- The RL binary method requires three registers: M, C, and P. However, we note that P can be used in place of M, if the value of M is not needed thereafter.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

- The multiplication (Step 2a) and squaring (Step 2b) operations in the RL binary method are independent of one another, and thus these steps can be parallelized. Provided that we have two multipliers (one multiplier and one squarer) available, the running time of the RL binary method is bounded by the total time required for computing h-1 squaring operations on k-bit integers.

As we can see from the *binary algorithm* above, modular multiplication is the basic operation used for modular exponentiation. This operation will be expounded in the following section.

2.2.4 Modular Multiplication Operation

The modular multiplication problem is defined as the computation of $P = AB \pmod{N}$ given the integers A, B, and N. It is usually assumed that A and B are positive integers with $0 \le A,B < N$, i.e., they are the least positive residues. There are basically four approaches for computing the product P.

- Multiply and then divide.
- The steps of the multiplication and reduction are interleaved.
- Brickell's method.
- Montgomery's method.

The multiply-and-divide method first multiplies A and B to obtain the $2k$-bit number

$$P' := AB\ .$$

Then, the result P' is divided (reduced) by n to obtain the k-bit number

$$P := P' \,\%\, N\ .$$

We will not study the multiply-and-divide method in detail since the interleaving method is more suitable and also more efficient. The multiply-and-divide method is useful only when one needs the product P'.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

2.2.4.1 Interleaving Multiplication and Reduction

Let A_i and B_i be the bits of the k-bit positive integers A and B, respectively. The product P' can be written as

$$P' = A * B = A * \sum_{i=0}^{k-1} B_i 2^i = \sum_{i=0}^{k-1} (A * B_i) 2^i$$

$$= 2(\dots 2(2(0 + A * B_{k-1}) + A * B_{k-2}) + \dots) + A * B_0$$

This formulation yields the shift-add multiplication algorithm. We also reduce the partial product modulo N at each step:

1. $P := 0$
2. **for** i = 0 **to** k _ 1
   - 2a. $P := 2P + A * B_{k-1-i}$
   - 2b. $P := P \bmod N$
3. **return** P

Assuming that $A,B, P < N$, we have

$$P := 2P + A * B_j$$

$$\leq 2(N-1) + (N-1) = 3N-3$$

Thus, the new P will be in the range $0 \leq P \leq 3N-3$, and at most 2 subtractions are needed to reduce P to the range $0 \leq P < N$. We can use the following algorithm to bring P back to this range:

$$P' := P-N; \text{ If } P' \geq 0 \text{ then } P = P'$$

$$P' := P-N; \text{ If } P' \geq 0 \text{ then } P = P'$$

The computation of P requires k steps, at each step we perform the following operations:

- A left shift: $2P$
- A partial product generation: $A * B_j$

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

- An addition: $P := 2P + A * B_j$
- At most 2 subtractions:

  $P' := P\text{-}N$; If $P' \geq 0$ then $P = P'$

  $P' := P\text{-}N$; If $P' \geq 0$ then $P = P'$

The left shift operation is easily performed by wiring. The partial products, on the other hand, are generated using an array of AND gates. The most crucial operations are the addition and subtraction operations: they need to be performed fast. We have the following avenues to explore:

1) We can use the carry propagate adder, introducing $O(k)$ delay per step.
2) We can use the carry save adder, introducing only O(1) delay per step. However, sign information is not immediately available in the CSA. We need to perform fast sign detection in order to determine whether the partial product needs to be reduced modulo N.

2.2.4.2 Montgomery's Algorithm

The Montgomery algorithm computes

$$\text{MontMult}(A,B) = A * B * r^{-1} \bmod N$$

given $A,B < N$ and r such that $\gcd(N, r) = 1$ (N and r are coprime, which means that the greatest common divisor between these two numbers is 1). Even though the algorithm works for any r which is relatively prime to N, it is more useful when r is taken to be a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors. In this section, we introduce an efficient binary add-shift algorithm for computing MontMult(A,B), and then generalize it to the m-ary method. We take $r = 2^l$ where l is the length of N, and assume that the number of bits in A or B is

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

less than N. Let $A = (A_{l-1} A_{n-2} \cdots A_0)$ be the binary representation of A. The above product can be written as

$$2^{-l} \cdot (A_{l-1} A_{l-2} \cdots A_0) \cdot B = 2^{-l} \cdot \sum_{i=0}^{l-1} A_i \cdot 2^i \cdot B \pmod N$$

The product $t = (A_0 + A_1 \cdot 2 + \cdots + A_{l-1} \cdot 2^{l-1}) \cdot B$ can be computed by starting from the most significant bit, and then proceeding to the least significant, as follows:

1. $t := 0$
2. for $i = l - 1$ to 0
2a. $t := t + A_i \cdot B$
2b. $t := 2 \cdot t$

The shift factor 2^{-l} in $2^{-l} \cdot A \cdot B$ reverses the direction of summation. Since

$$2^{-l} \cdot (A_0 + A_1 \cdot 2 + \cdots + A_{l-1} \cdot 2^{l-1}) = A_{l-1} \cdot 2^{-1} + A_{l-2} \cdot 2^{-2} + \cdots + A_0 \cdot 2^{-l},$$

we start processing the bits of A from the least significant, and obtain the following binary add-shift algorithm to compute $t = A \cdot B \cdot 2^{-l}$.

1. $t := 0$
2. for $i = 0$ to $l - 1$
2a. $t := t + A_i \cdot B$
2b. $t := t/2$

The above summation computes the product $t = 2^{-l} \cdot A \cdot B$, however, we are interested in computing $u = 2^{-l} \cdot A \cdot B \pmod N$. This can be achieved by subtracting N during every add-shift step, but there is a simpler way: We add N to u if u is odd, making new u an even number since N is always odd. If u is even after the addition step, it is left untouched. Thus, u will always be even before the shift step, and we can compute

$$u := u \cdot 2^{-1} \pmod N$$

by shifting the even number u to the right since $u = 2v$ implies

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

$$u := 2v \cdot 2^{-1} = v \pmod{N}$$

The binary add-shift algorithm computes the product $u = 2^{-n} \cdot A \cdot B \pmod{N}$ as follows:

1. $u := 0$
2. for $i = 0$ to $l - 1$

2a. $u := u + A_i \cdot B$

2b. If u is odd then $u := u + N$

2c. $u := u/2$

We reserve a $(l + 1)$-bit register for u because if u has k bits at beginning of an add-shift step, the addition of $A_i \cdot B$ and N (both of which are l-bit numbers) increases its length to $l + 1$ bits. The right shift operation then brings it back to l bits. After k add-shift steps, we subtract N from u if it is larger than N. Also note that Steps 2a and 2b of the above algorithm can be combined: We can compute the least significant bit u_0 of u before actually computing the sum in Step 2a. It is given as

$$u_0 := u_0 \oplus (A_i \cdot B_0) .$$

Thus, we decide whether u is odd prior to performing the full addition operation, $u := u + (A_i \cdot B)$. This is the most important property of Montgomery's method. In contrast, the classical modular multiplication algorithm (e.g., the interleaving method) computes the entire sum in order to decide whether a reduction needs to be performed.

2.3 Field Programmable Gate Arrays (FPGAs)

To achieve optimal system performance while maintaining physical security, it is desirable to implement cryptographic algorithms in hardware. Many public-key cryptographic algorithms require the implementation of modular arithmetic, specifically modular multiplication, for operands of 1024 bits in length such as RSA. Additionally, algorithm

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

agility is required to support algorithm independent protocols, a feature of most modern security protocols. Reprogrammability, particularly in-system reprogrammability, is critical in enabling the switching between cryptographic algorithms required for algorithm independent protocols [12].

Field Programmable Gate Arrays (FPGAs) implementation makes it feasible for these algorithms to be reprogrammed. FPGAs are highly attractive for the continuous evolution of cryptographic algorithms and for the new algorithms created to meet security needs. The target FPGA should be designed with the architectural requirements for wide-operand modular arithmetic in mind in an effort to maximize system performance.

The computational complexity of many public-key algorithms is dependent on the system's ability to perform modular addition. Redundant Representation and Systolic Array implementations are two viable methods for carrying out high performance modular addition [12]. For this research, systolic array implementation method has been chosen for the architectural design of RSA.

2.3.1 FPGA Technologies and Their Suitability for Modular Arithmetic

FPGAs are normally configured based on SRAM, EPROM, EEPROM, or antifuse technology. Antifuse based FPGAs may only be programmed once while EPROM-based and EEPROM-based FPGAs retain their programming data even after power is removed and may be electrically reprogrammed. However, reprogramming EPROM-based and EEPROM-based FPGAs requires high voltages and therefore is not typically done while the devices are in-system. SRAM-based FPGAs are fully in-system reprogrammable, though they must be reprogrammed each time the system is powered-up, typically from a ROM device containing the FPGA's configuration data. Due to their in-system reprogrammability, SRAM-based FPGAs offer the greatest amount of implementation

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

flexibility and the architectural examination will focus on these types of FPGAs. The most popular SRAM-based FPGAs are the Xilinx FPGA, the Altera FPGA, and the Lucent FPGA. Based from the analysis done by Elbirt and Paar [12], Xilinx and Lucent FPGAs are desirable for systolic array implementations, and for this research, Xilinx FPGA will be considered.

As it is desirable to utilize reprogrammable devices such as FPGAs for the implementation of cryptographic algorithms, a great deal of knowledge may be gained by examining current FPGA architectures to determine their strengths and weaknesses when used to perform arithmetic operations.

2.3.1.1 Xilinx FPGA Architectural Analysis

Xilinx FPGAs are comprised of a two-dimensional array of configurable logic blocks (CLBs) with horizontal and vertical routing channels used to interconnect the CLBs. The CLBs contain look-up-tables (LUTs) and flip-flops. The look-up-tables may be configured as either combinatorial logic or as RAM. Additionally, each CLB contains circuitry to implement fast carry operations for arithmetic circuits. Hard-wired carry logic exists within each CLB to both accelerate and condense arithmetic functions. The carry logic and function generators share operand and control inputs. Dedicated high-speed routing channels are used to route the ripple-carry outputs between CLBs.

The CLB's fast ripple-carry propagates along paths that run left to right for the top and bottom rows and both horizontally and vertically for all columns. This feature is critical for Systolic Array modular multiplication implementations when the size of the processing element is larger than two bits, exceeding the size of a single CLB.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

2.3.1.2 FPGA Architecture Consideration for Public-Key Algorithm Implementation

Xilinx FPGAs are desirable when implementing Systolic Array modular multiplication implementations. Here are some important considerations and techniques we should put in mind when implementing our design through FPGA.

- A two-dimensional array of configurable units interconnected via horizontal and vertical routing channels yields the greatest flexibility when performing design placement and routing.
- Fast ripple-carry routing paths have been shown to be critical to achieve high performance addition and allowing these high-speed routing paths to connect to any of the adjacent configurable units, yields greater placement flexibility of implementation elements.
- When creating the configurable unit, a look-up-table format capable of computing multiple arithmetic bits within one unit will minimize ripple-carry propagation between units and maximize the utilization of each unit.
- The look-up-tables must be interconnected using high-speed carry logic to maximize the performance of arithmetic operations with the configurable unit.
- Finally, minimizing the number of ripple-carries between adjacent configurable units is a key factor in increasing the performance of a wide-operand addition.

Increasing the number of arithmetic bits that may be computed within one configurable unit results in a decrease in the number of configurable units required for a wide-operand addition. While this will serve to increase system performance, the associated cost is an increase in the unit-delay of the configurable unit. The break-even point occurs when the unit-delay of the configurable unit is equivalent to the delay in routing ripple-

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

carries to adjacent units via the fast ripple carry routing paths. Therefore, the number of arithmetic bits that may be computed within one configurable unit must be carefully chosen based on the timing associated with the fast ripple-carry routing paths.

In Xilinx XC4000 FPGA, the carry delay of a sixteen-bit adder is approximately equivalent to one CLB delay in a. It can therefore be deduced that a configurable unit can compute sixteen arithmetic bits in approximately the same amount of time required to propagate the associated ripple-carry output to an adjacent configurable unit. This architecture results in an n-bit addition requiring $n/16$ configurable units and $2n/16 - 1$ unit delays.

Combining all of the aforementioned techniques leads to a theoretical FPGA architecture that will maximize the performance of high-speed addition algorithms. This will result in a performance increase for Systolic Array implementations of modular multiplication and, as a result, cryptographic algorithms.

King Mongkut's Institute of Technology Ladkrabang

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Chapter 3

Previous Works on Modular Multiplication

In this chapter, the mathematical overview that is extensively used in cryptographic algorithms will be introduced. Several of the existing systolic array architecture designs and FPGA implementations will be discussed.

3.1 Modular Arithmetic

As discussed in the previous chapter 2, IDEA and RSA algorithm are based on modular arithmetic. This section will present the properties of modular arithmetic which can be useful in algorithm development that are being implemented in RSA and IDEA.

Lemma 2.1.1 $i \bmod n = (i + kn) \bmod n$ *for any integer k.*

Proof: If $i = nq + r$, with $0 \le r < n$, then $i+kn = n(q+k)+r$, so by definition $r = i \bmod n$ and $r = (i + kn) \bmod n$.

Lemma 2.1.2

$$
\begin{aligned}
(i + j) \bmod n &= [i + (j \bmod n)] \bmod n \\
&= [(i \bmod n) + j] \bmod n \\
&= [(i \bmod n) + (j \bmod n)] \bmod n \\
(i * j) \bmod n &= [i * (j \bmod n)] \bmod n \\
&= [(i \bmod n) * j] \bmod n \\
&= [(i \bmod n) * (j \bmod n)] \bmod n
\end{aligned}
$$

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Proof: We prove the first and last terms in the sequence of equations for plus are equal; the other equalities for plus follow appropriate substitutions or by similar computations, whichever you prefer. The proofs of the equalities for products are similar. We use the fact that $j = (j \bmod n) + nk$ for some integer k and that $i = (i \bmod n) + nm$ for some integer m. Then

$$(i + j) \bmod n = [(i \bmod n) + nm + (j \bmod n) + nk)] \bmod n$$
$$= [(i \bmod n) + (j \bmod n) + n(m + k)] \bmod n$$
$$= [(i \bmod n) + (j \bmod n)] \bmod n$$

The functions used in encryption and decryption in the RSA system use a special arithmetic on the numbers 0, 1, . . . , n-1.

Theorem 2.1.3 *Addition and multiplication mod n satisfy the commutative and associative laws, and multiplication distributes over addition.*

Proof: Commutativity follows immediately from the definition and the commutativity of ordinary addition and multiplication. We prove the associative law for addition below, the other laws follow similarly.

$$a +_n (b +_n c) = (a + ((b + c) \bmod n)) \bmod n$$
$$= (a + b + c) \bmod n$$
$$= ((a + b) \bmod n + c) \bmod n$$

Notice that $0 +_n i = i$, $1 *_n i = i$, and $0 *_n i = 0$, so we can use 0 and 1 in algebraic expressions mod n as we use them in ordinary algebraic expressions.

We conclude this section by observing that repeated applications of Lemma 2.1.2 are useful when computing sums or products in which the numbers are large. For example, suppose you had m integers $x1, \ldots, xm$ and you wanted to compute $(\sum_{j=1}^{m} x_i) \bmod m$.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

One natural way to do so would be to compute the sum, and take the result modulo m. However, it is possible that, on the computer that you are using, even though $(\sum_{j=1}^{m} x_i)$ mod m is a number that can be stored in an integer, and each xi can be stored in an integer, $\sum_{j=1}^{m} x_i$ might be too large to be stored in an integer. Lemma 2.1.2 tells us that if we are computing a result mod n, we may do all our calculations in Z_n using modular addition and modular multiplication, and thus never computing an integer that has significantly more digits than any of the numbers we are working with.

3.2 Modular Multiplication

The previous works in modular multiplication utilizing systolic array will be presented in this section. The motivation of this research is to design a better approach and compare to one of the approaches [3] discussed in section 3.2.2.

3.2.1 Systolic Array Architectures [2, 6]

C. D. Walter [2] proposed systolic modular multiplication utilizing Montgomery's method that led to various implementation and different techniques on how to further improve its efficiency [3, 4, 6, 18].

The proposed design is two-dimensional (mxm) systolic array for modular multiplication using the algorithm of P. L. Montgomery. The throughput is one modular multiplication every clock cycle with latency of $2n$+2 cycles for multiplicands having n digits. Each row of the array performs the iteration of the loop and columns compute successive values for a single bit position. The typical cell performs a single digit.

P. Kornerup [6] describes an architecture based on one row of processing elements and a radix of two. For exponentiation computation, squarings and multiplications are computed in parallel. Each cell produces two digit-product terms and accumulates these

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

into a previous sum of the same weight, developing the product least significant digit first. Through this approach, only $\lceil n/2 \rceil$ cells are needed for a full n x n multiplication. Two such multipliers interconnect to form a purely systolic modulo exponentiator, thus the system requires n systolic processing elements for an n-bit modular exponentiation and the resulting execution time is $2n^2$ clock cycles.

3.2.2 Different Architectures and FPGA Implementations

This section will discuss various architectures for modular multiplication. The trade-offs between sequential and parallel implementation will be shown as well as the advantage and disadvantage of efficient architecture. Lastly, FPGA-dependent architecture will be presented.

3.2.2.1 Sequential vs. Parallel Hardware Implementations [3]

N. Nedjah and L. M. Mourelle [3] proposed another systolic-based architecture that was synthesized and implemented in FPGA with a latency of $2n+3$ clock cycles. The synthesis was done for Virtex-E family and the implementation device used is a SPARTAN, model S05PC84-4. The author formulated a systolic algorithm derived from the modified Montgomery algorithm.

The first algorithm below (modified Montgomery algorithm) was further modified (2nd algorithm) to make it more suitable to systolic implementation. It can be seen from the second algorithm that the value of new R depends on the values of a_i and q_i, hence, the computation of $R + a_iB + q_iM$ can be tabulated as shown in Table 3.1.

Algorithm 3.1 Montgomery modular algorithm [3]

Montgomery(A, B, M) {
int R = 0;
1. **for** i= 0 to n-1
2. { $R = R + a_iB$
3. **if** $r_0 = 0$ **then**

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
4               R = R div 2
5.         else
6.              R = (R + M) div 2;
       }
return R;
```

Algorithm 3.2 Modified Montgomery modular algorithm [3]

```
ModifiedMontgomery(A, B, M) {
int R = 0;
1.  for i = 0 to n-1
2.  {   q_i = (r_o + a_i b_o) mod 2;
3.      R = (R + a_i B + q_i M) / 2;
     }
return R;
```

Table 3.1 Computation of $R + a_iB + q_iM$ (MB = M+B, precomputed)[3].

A_i	q_i	$R + a_iB + q_iM$
1	*1*	$R + MB$
1	*0*	$R + B$
0	*1*	$R + M$
0	*0*	R

Algorithm 3.3 Systolic Montgomery modular algorithm [3]

```
SystolicMontgomery (A,B,M,MB)
{       int R =0; bit carry = 0, x;
        0.  for i = 0 to n
        1.  {   q_i = r_(i),0 ⊕ a_i *b0;
        2.       for j = 0 to n
        3.       {       switch a_i, q_i {
        4.                       1,1:  x = mb_i;
        5.                       1,0 :  x =b_i ;
        6.                       0,1 :  x = m_i ;
        7.                       0,0 :  x = 0 ;
                 }
        8.       r_(i+1),0 = r_(i),(j+1) ⊕ x_i ⊕ carry;
        9.       carry = r_(i),(j+1) * x_i + r_(i),(j+1) * carry + x_i * carry;
             }
        return R;
}
```

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

The systolic architecture of the systolic Montgomery multiplier in [3] is shown in Figure 3.1. All PEs perform the same task except that the right-border PEs have to compute bit q_i as well, line 1 of systolic algorithm above.

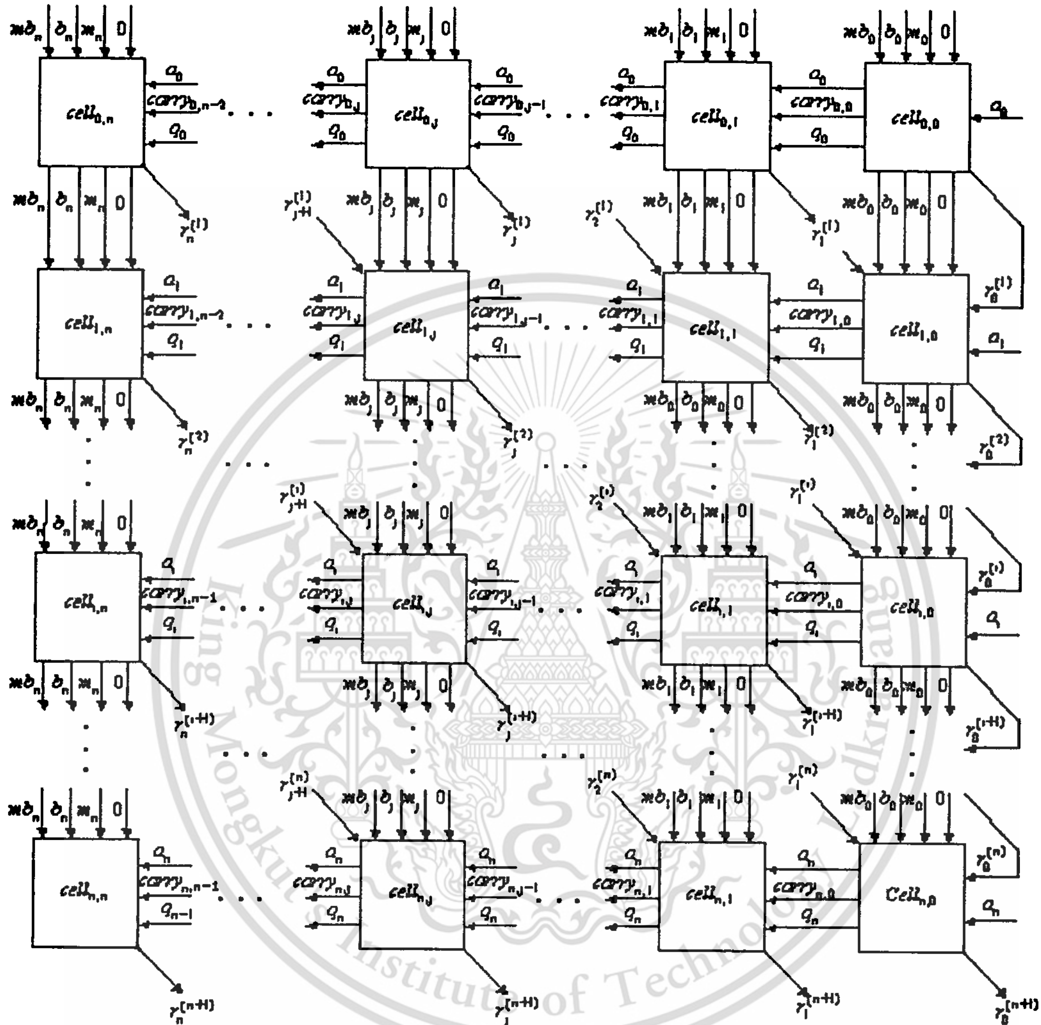


Figure 3.1 Systolic architecture of Montgomery multiplier [3].

The basic processing element is shown in Figure 3.2, while Figure 3.3 shows the left border PEs wherein $r_{(i),n} = 0$.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

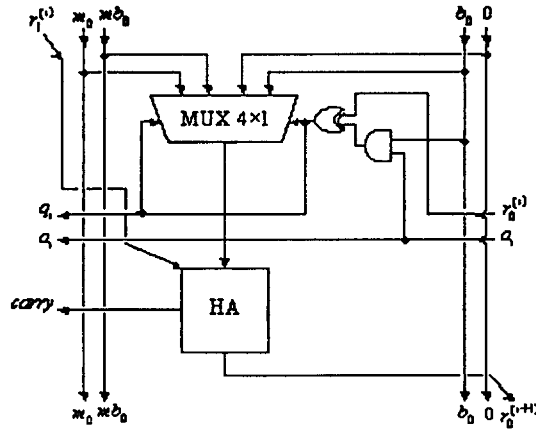


Figure 3.2 Right border PEs of mxm array [3].

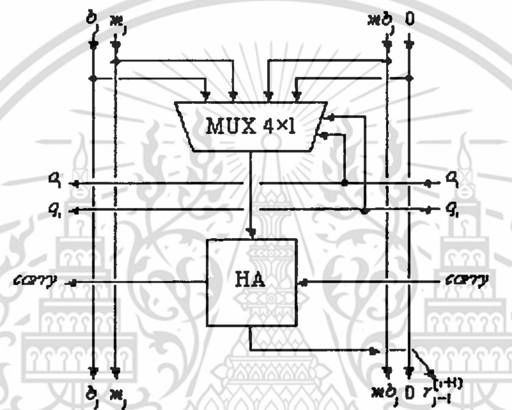


Figure 3.3 Top-most PEs of mxm array [3].

The author compared this systolic multiplier (SM) with their iterative multiplier (IM), and as expected, iterative modular multiplier reduces the required hardware area at the expense of response time. The systolic implementation attempts to minimize time requirements at the expense of hardware area. Table 3.2 shows the computation time, area and *area-time* product resulted from the synthesis result.

Table 3.2 Iterative vs. systolic Montgomery modular multiplier [3].

Operand size	Area (CLBs)		clock cycle time (ns) (Execution Time)		Area x time	
	IM	SM	IM	SM	IM	SM
128	89	259	46	23	4094	5957
256	124	304	102	42	12648	12767
512	209	492	199	76	41591	37392
768	335	57	207	82	69345	47396

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

1024	441	639	324	134	142884	85626

This systolic Montgomery multiplier was implemented in binary modular exponentiation by the same authors [4].

3.2.2.2 Resource vs. Speed Efficient Architecture [1]

From the master thesis of Blum [1], two designs have been done: First is the ***resource efficient architecture***, also presented in [7]. RSA was implemented in one row of processing elements. With this approach two modular multiplications can be processed simultaneously and the performance reduces to a throughput of two modular multiplications per $2n$ cycles. The latency is $2m$ cycles. B+M is pre-computed, to avoid 2-bit carry as what the authors in reference [3] did. Each processing element computes u bits of a modular multiplication where u = 4, 8, 16. Algorithm 3.4, is a new version of Montgomery algorithm implemented in this design using radix r =2. Figure 3.4 shows the processing element that computes line 4 of Algorithm 3.4 where $q_i, a_i \in \{0, 1\}$.

Algorithm 3.4 Another version of Montgomery's Algorithm [1]

1. $S_0 = 0$
2. **for** $i = 0$ to $m + 2$ do
3. $q_i = S_i \bmod 2$
4. $S_{i+1} = (S_i + q_i * M)/2 + a_i * B$
5. **end** for

Part of the processing elements is the following registers:

- M-Reg (u bits): storage of the modulus
- B-Reg (u bits): storage of the B multiplier
- B+M-Reg (u bits): storage of the intermediate result B +M
- S-Reg (u + 1 bits): storage of the intermediate result (inclusive carry)
- S-Reg-2 (u − 1 bits): storage of the intermediate result
- Control-Reg (3 bits): control of the multiplexers and clock enables

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

- q_i, a_i (2 bits) : multiplier A, quotient Q
- Result-Reg (u bits): storage of the result at the end of a multiplication

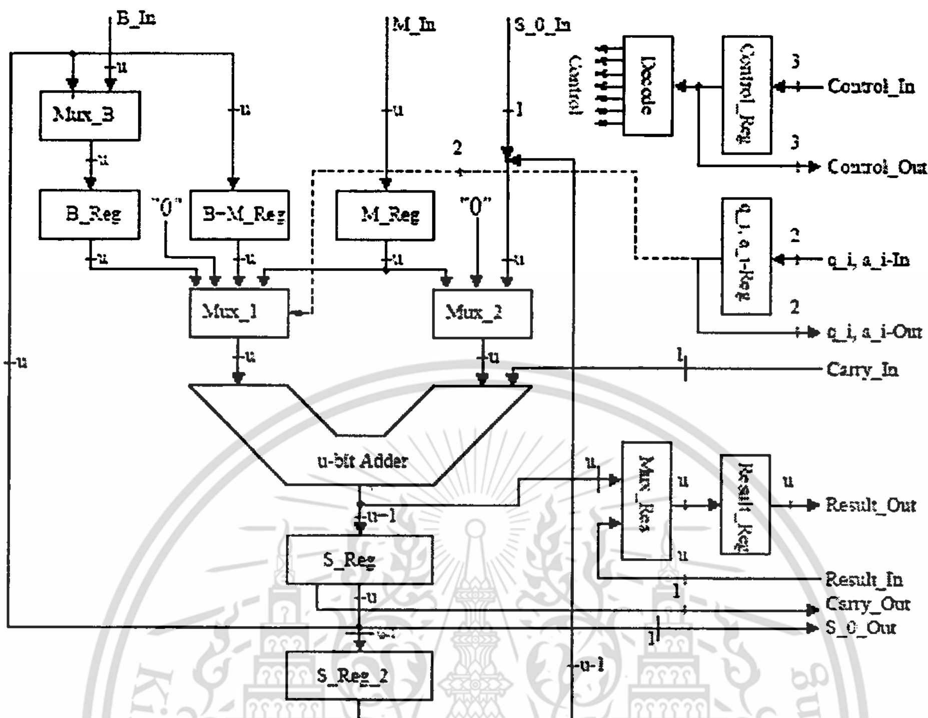


Figure 3.4 Processing Element (unit) of resource efficient architecture [1].

This design was implemented for various bit lengths and unit widths. Table 3.3 shows the results in terms of used CLBs (A), clock cycle time (T) and the time-area product (TA). Table 3.4 shows the application of the results to public-key schemes where the Chinese remainder theorem cannot be applied. A full modular exponentiation with an n bit exponent is computed in $2(n+1)(n+4)$ clock cycles. Table5 shows the result when the design is applied to RSA using F_4 expoent $(2^{16}+1)$, which is 17 bits long, the encryption thus required $2*19(n+4)$ clock cycles. The CLB consumption is approximately $3u+4$ CLBs per unit. The time-area product shows that designs with 8-bit units are generally most efficient.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

This design has two advantages as compared to previous implementations, 1) this design stores the modulus, the exponent and the pre-computation factor in registers and RAM; 2) this design is implemented into one device.

Table 3.3 CLB usage, minimal clock cycle time, and time-area product of modular exponentiation architectures on Xilinx FPGAs[1].

U	256 bit			512 bit			768 bit			1024 bit		
	Area (CLBs)	T (ns)	TA (CLBs*ns)	Area (CLBs)	T (ns)	TA (CLBs*ns)	Area (CLBs)	T (ns)	TA (CLBs*ns)	Area (CLBs)	T (ns)	TA (CLBs*ns)
4	1307	17.5	22875	2555	17.7	45223	3745	19.1	71529	4865	19.2	93408
8	1122	19.8	22215	2094	19.1	39995	3132	19.4	60760	4224	23.4	98842
16	1110	21.7	23870	2001	21.8	43621	2946	21.6	63633	3786	23.7	89728

Table 3.4 CLB usage and execution time for a full modular exponentiation[1]

	512-bit		768-bit		1024-bit	
u	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)
4	2555	9.38	3745	22.71	4865	40.50
8	2094	10.13	3123	23.06	4225	49.36
16	2001	11.56	2946	25.68	3786	49.78

Table 3.5 Application to RSA: Encryption [1]

	512-bit		1024-bit	
u	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)
4	2555	0.35	4865	0.75
8	2094	0.37	4225	0.91
16	2001	0.43	3786	0.93

To speed-up the design two approaches can be taken, a) reduce the cycle time, or b) reduce the number of cycles/steps per modular multiplication. Reduction of the cycle time can be achieved by computing one instead of u bits per unit. Compared to a design with 4 bit units, the resulting speed-up of approximately 20% (without three ripple carry delays)

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

comes at the expense of additional 30% resources. Thus the time-area product becomes worse.

In the second design, the second approach mentioned is implemented, the *speed efficient architecture*, also presented in [5], Using a radix $r = 2^k$, $k > 1$, reduces the number of steps by a factor k.

Each processing element (PE) in Figure 3.5 computes 4 bits of a modular multiplication. The registers need a total of 14 CLBs, the adders 13 CLBs and the RAM blocks 4 CLBs. The possibility of reusing registers for combinatorial logic allows some savings of CLBs. Thus a processing element utilizes a total of 24 CLBs, which is equal to 6 CLBs per processed bit.

The time-area products of Table 3.6 are between 50% and 70% larger compared to those of Table 3.3 but far more efficient than *resource efficient architecture*, the speed-up is approximately a factor 4 by computing a modular exponentiation with 4 times fewer cycles. Table 3.5 shows the CLB usage and computation time. A full modular exponentiation with an n bit exponent and an m digit modulus is computed in $2\ (n+2)(m+10)$ clock cycles. Table 3.7 shows the comparison between this design and Design 1, where $u = 4$, which is the fastest in Table 3.4. The speed-up is approximately 3.5. Table 3.8 shows the comparison between Design 1 and Design 2 when applied to RSA encryption.

Table 3.6 CLB usage(Area), minimal clock cycle time(T), and time-area(TA) product of modular exponentiation architectures on Xilinx FPGAs [1].

256 bit			512 bit			768 bit			1024 bit		
Area (CLBs)	T (ns)	TA (CLBs*μs)	Area (CLBs)	T (ns)	TA (CLBs*μs)	Area (CLBs)	T (ns)	TA (CLBs*μs)	Area (CLBs)	T (ns)	TA (CLBs*μs)
1818	21.3	38.7	3413	20.7	70.6	5071	20.1	101.9	6633	21.9	145.2

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

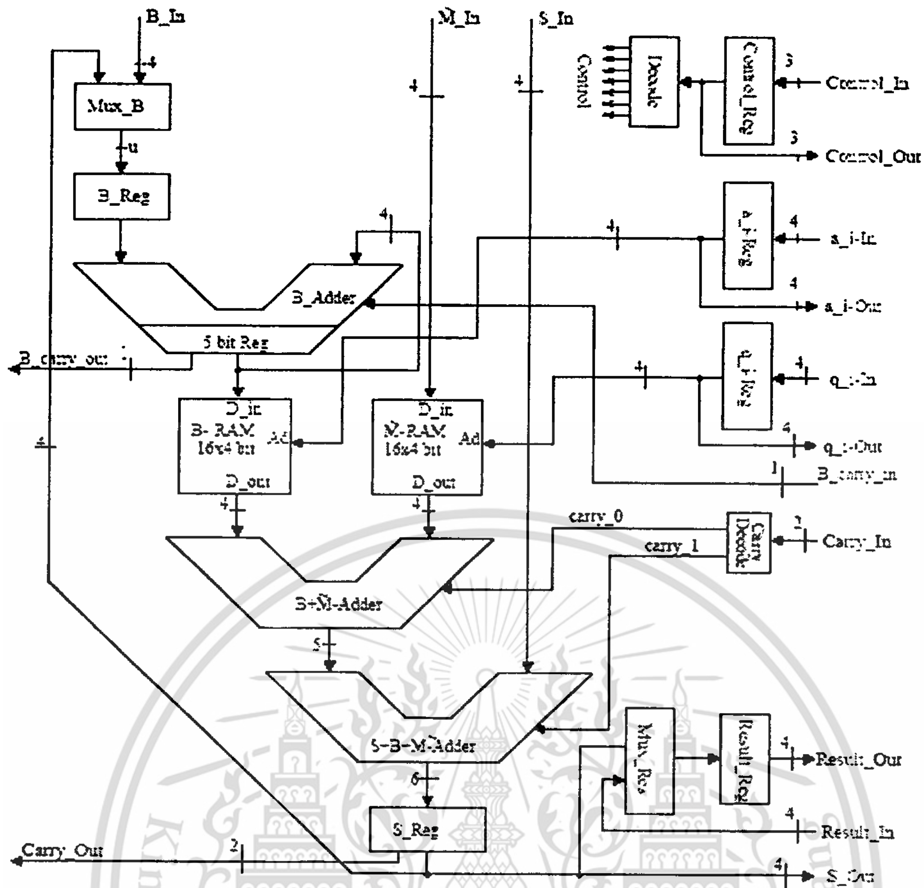


Figure 3.5 Processing Element (unit) of speed efficient architecture [1].

Table 3.7 CLB usage and execution time for a full modular exponentiation of design 1(where u=4) and design 2 [1].

DESIGN	512-bit		768-bit		1024-bit	
	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)
1	2555	9.38	3745	22.71	4865	40.50
2	3413	2.93	5071	6.25	6633	11.95

Table 3.8 Application to RSA: Encryption[1]

		512-bit		1024-bit	
Design	*u*	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)
1	4	2555	0.35	4865	0.75
	8	2094	0.37	4224	0.91
	16	2001	0.43	3786	0.93
2		3413	0.11	6633	0.22

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

3.2.2.3 FPGA Architecture-Dependent Design [8]

Daly's design [8], takes into account the underlying architecture of the target technology, specifically for reconfigurable device. In this design, the calculation is divided into j p-bit words, pipelined it, through this technique, an improvement in clock speed is achieved, p is the maximum length of the carry chain for the device and $j = n/p$. Algorithm 3.5 is another version of Montgomery's algorithm that is implemented for this design. The processing element consists of multiplexer and adder, shown in Figure 3.6.

Algorithm 3.5 Another version of Montgomery's Algorithm [8]

```
MonPro (A,B,M)
{       S-1 = 0 ;
        A = 2 x A ;
        for i = 0 to n do
                qi := (Si-1) Mod 2 ;     // (LSB of Si-1)
                Si = (Si-1 + qiM + biA)/2;
        end for;
        return Sn ;
}
```

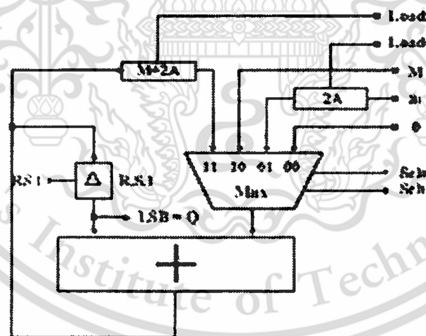


Figure 3.6 Montgomery modular multiplier architecture with single adder and multiplexer [8].

The input to the multiplier is broken into p-bit words, and the carry out of the adder is delayed by one clock cycle before being inputted to the next adder. However, due to the right-shift operation, the LSB of the sum must be fed *directly* into the MSB of the *previous* adder as illustrated in Figure 3.7. This does not slow the operation of the circuit since the

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

LSB of the higher adder will be computed long before the MSB of the lower adder needs to be computed.

Figure 3.7 Pipeline modular multiplier [8]

Figure 3.8 illustrates the flow of data through a fully pipelined multiplier with $j = 4$. It can be seen that there is a delay of n+3 clock cycles between the loading of the first p-bit input word, A_0 and the first p-bit output word, S_0 being valid in the result register of the same pipelined unit. It takes n+3 clock cycles for each multiplication due to the 2 clock cycle initialization (loading the RAM with the values of 2A, M, and then M+2A), and the n+1 clock cycle implementation of Algorithm 3.5. For the next j-1 clock cycles, output words S_1-S_{j-1} are valid and are right shifted as they are being loaded so as to output the result serially over k clock cycles.

It is obvious that another multiplication can begin as soon as the first output word is valid, and has been latched into the result register. Refer to Figure 3.8 where each of the blocks represents a single pipelined multiplier unit as in Figure 3.7. Although it takes (n+3)+(j-1) clock cycles from the input of A_0 to the latching of S_{j-1} into the result register, a new result is valid every (n + 3) clock cycles thereafter. Therefore for throughput calculation purposes, it takes (n+3) clock cycles to produce the consecutive multiplication results, with an extra latency of (j -1) clock cycles to produce the first result. This latency will not affect the throughput of the exponentiator.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

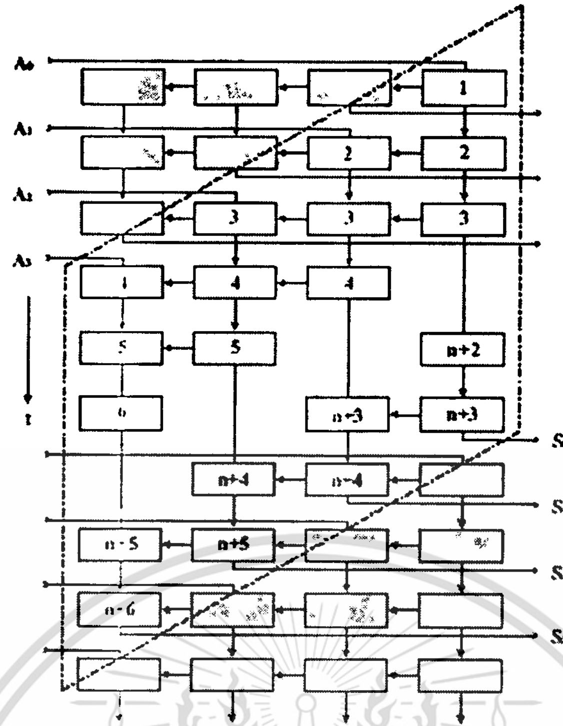


Figure 3.8 Pipelined Multiplication: DDG using the pipelined multiplier units [8].

The target device for this design was the Xilinx Virtex V1000FG680-6, which has 124 CLB's per column. Table 3.9 shows the speed and area comparisons for pipelined multipliers up to 1080 bits. Therefore p (the bit length of the pipelined multiplier unit) was chosen to be 120, allowing for space for control registers needed for each unit. So for the 1080-bit design, $p = 120$, $j = 9$, that is, 9 of the pipelined units were required for each multiplier. The design also included all the registers necessary for using the multiplier in the exponentiator design. Therefore, the designs occupied more area than would be required for a single purpose multiplier. The significant decrease in clock frequency when the bitlength is increased from 120-bits to 240 is due to the routing of the carry from the top of one column in the FPGA to the bottom of the next. However, this penalty is not as great as for the non-pipelined design as can be seen in Table 3.10, and the frequency does not continue to decrease dramatically. Approximately 2.5 CLB's per bit of the multiplier are required for this design.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table 3.10 provides an indication as to how effective the pipelining of the multiplier is. The design used to produce the results in Table 3.10 was identical to one multiplier unit, except the bit length and thus carry chain length were increased to 240, 480, 720 and 1080-bits. A dramatic decrease in clock frequency is observed as the carry chain exceeds the column height of the FPGA. Thus it is obvious that the pipelined multiplier out-performs the non-pipelined design provided that a suitable pipeline bit-length is chosen to exploit the maximum length of the carry chain. Finally, a full modular exponentiator is implemented and synthesized, and the results are presented in Table 3.11. This design can implement either RSA encryption or decryption.

Table 3.9 Speed & area results for the pipelined Mongtomery modular multiplier[8].

Multiplier Size	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	603 (4%)	88.47	84.9 Mb/s
240	1211 (9%)	57.99	56.8 Mb/s
480	2426 (19%)	59.03	58.4 Mb/s
720	3641 (29%)	56.83	56.4 Mb/s
1080	5458 (44%)	54.61	54.4 Mb/s

Table 3.10 Speed & area results for a non-pipelined Mongtomery modular multiplier[8].

Multiplier Size	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	603 (4%)	88.47	84.9 Mb/s
240	1203 (9%)	41.41	40.6 Mb/s
480	2403 (19%)	18.52	18.3 Mb/s
720	3603 (29%)	12.64	12.6 Mb/s
1080	5403 (43%)	8.40	8.4 Mb/s

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table 3.11 Speed & area results for the pipelined RSA encryptor/decryptor [8].

Bit Length (n)	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	1146 (9%)	83.51	673.2 kb/s
240	2301 (18%)	58.15	238.3 kb/s
480	4610 (37%)	55.92	115.5 kb/s
720	6917 (56%)	50.66	70.0 kb/s
1080 (1024)	10369 (84%)	49.63	45.8 kb/s

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Chapter 4

Systolic Design Methodology

As mentioned in Chapter 2, Montgomery Algorithm is an efficient algorithm in performing modular multiplication for RSA, and in Chapter 3 different existing architectures and implementation of this algorithm is presented, this section will discuss a new technique that allow efficient and fast computation of modular multiplication operation.

4.1 Montgomery Modular Multiplication Algorithm

The Montgomery algorithm (Algorithm 4.1) [38] shows how to perform modular multiplication by a method of reversing the order of the multiplicand, shifting down instead of shifting up, and adding rather than subtracting multiples of the modulus. Here we are going to adopt the modified Montgomery algorithm (Algorithm 4.2) and Systolic Montgomery (Algorithm 4.3) of [3].

Let A, B and N be the multiplier, multiplicand and modulus respectively such that $0 \leq A, B < N$. The modulus should be relatively prime to the radix, in case of RSA and IDEA where the modulus is always odd, this condition is always true when the value of radix is two.

The representations of each operand are as follows: $A = (a_{l-1},\ldots, a_{1,} a_0)$, $B = (b_{l-1},\ldots, b_{1,} b_0)$ and $N = (n_{l-1},\ldots, n_{1,} n_0)$ where l is the length of the modulus and $a_i, b_i, n_i \in \{0,1\}$.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Algorithm 4.1 Radix-2 Montgomery multiplication algorithm

```
Montgomery(A, B, N)
{       P = 0;
1.      for i = 0 to l-1
2.      {       if (P + a_i B) is even
3.                      P = (P + a_i B) / 2;
4.              else
5.                      P = (P + a_i B + N) / 2;
        }
6.      if P ≥ N
7.              P = P - N;
        return P;
}
```

In Algorithm 4.2, it can be seen that line 3: $P = (P + a_iB + q_i N) / 2$, is dependent on the values of a_i and q_i, Table 4.1 summarizes its computation where precomputed $NB = (nb_{l-1},\ldots, nb_1, nb_0)$, is the sum of N and B, with $nb_i \in \{0,1\}$.

The final subtraction in line 7 is intentionally omitted, because the technique that will be discussed in the succeeding section will concentrate mainly on the iteration involved inside the *for*-loop of Systolic Montgomery algorithm.

Algorithm 4.2 Modified Montgomery multiplication algorithm

```
ModifiedMontgomery(A, B, N)
{       P = 0;
1.      for i = 0 to l-1
2.      {       q_i = (p_o + a_i b_o) mod 2;
3.              P = (P + a_i B + q_i N) / 2;
        }
        return P;
}
```

Table 4.1 Computation of $P + a_iB + q_iN$
(NB = N + B, precomputed)

a_i	q_i	$P + a_iB + q_i N$
1	1	$P + NB$
1	0	$P + B$
0	1	$P + N$
0	0	P

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Algorithm 4.3 is the modified version of Algorithm 4.2, which is more suitable to systolic implementation [3]. The corresponding dependence graph of Algorithm 4.3 and with the data flowing in it is shown in Figure 4.1. All PEs perform the same task except that the right-border PEs have to compute bit q_i as well, see line 2.

Algorithm 4.3 Systolic Montgomery modular algorithm

```
SystolicMontgomery (A, B, N, NB)
{       int P =0;
        bit carry = 0,  x;
        1.  for i = 0 to l
        2.  {   q_i = p_(i),0 ⊕ a_i * b_0;
        3.        for j = 0 to l
        4.        {        switch a_i, q_i
        5.                 {        1,1:  x  = nb_i;
        6.                          1,0:  x = b_i;
        7.                          0,1:  x  = n_i;
        8.                          0,0:  x  = 0;
                           }
        9.        p_(i+1),0 = p_(i),(j+1) ⊕ x_i ⊕ carry;
        10.       carry = p_(i),(j+1) * x_i  +  p_(i),(j+1) * carry  +  x_i * carry;
                  }
            }
        return P;
}
```

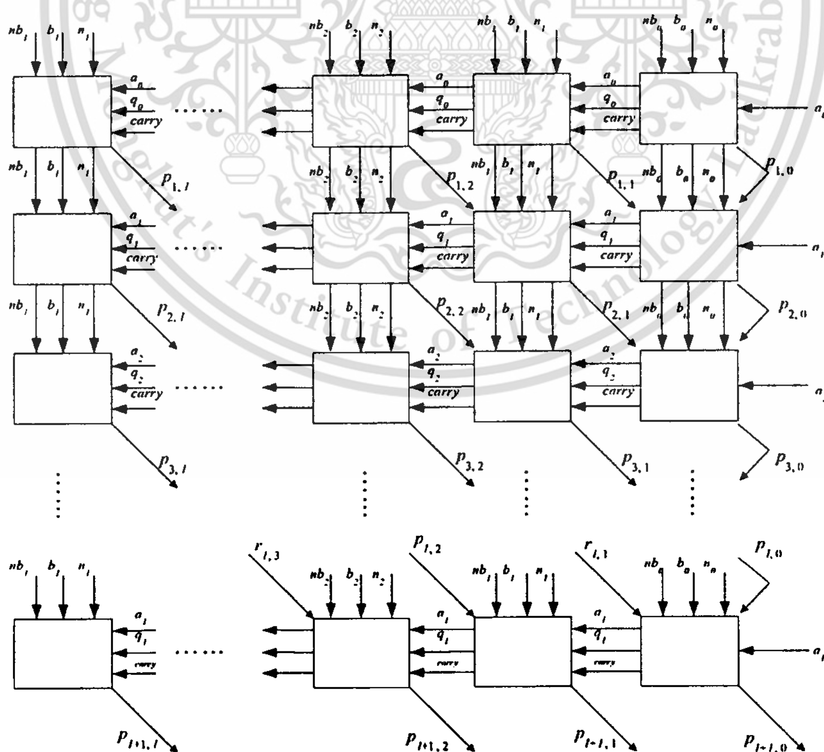


Figure 4.1 Dataflow within the Dependence Graph based on Algorithm 4.3

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

4.2 Mapping Procedure [13]

There are three design levels involved in transforming an algorithm to an array of processors;

- Stage 1: **DG (dependence graph) Design**. After identifying the specific algorithm, the next step is to generate a dependence graph for it. Here, the Algorithm 4.3 in Section 4.1 will be utilized for our design. Note that DG greatly affects the final array design [13], it is desirable to consider the best DG design before proceeding to the next stage.
- Stage 2: **SFG (Signal Flow Graph) Design**. Different SFGs can be derived from a given DG, it will be shown later that these SFGs have major impact on the final architecture as well as the performance in terms of computation and delay.
- Stage 3: **Processor Array Design**. The SFG can be physically realized in terms of SIMD, systolic, wavefront, or MIMD array. For this research, systolic design will be considered and will be the main focus of the discussion.

4.2.1 Algebraic Mapping Procedure

Given a DG of dimension n, a projection vector, $\vec{d}$, and a permissible linear schedule, $\vec{s}$, an SFG can be derived based on the following mappings:

1) *Node mapping*. This mapping assigns the node activities in the DG to processors. The mapping of computation **c** on the DG into a node $\boldsymbol{n}$ of SFG is:

$$\boldsymbol{n} = P^T \mathbf{c}$$

where the *processor basis P*, is orthogonal to $\vec{d}$; $P^T \vec{d} = 0$.

2) *Arc mapping*. This mapping maps the arcs of the DG to the edges of the SFG. The set of edges $\vec{e}$ into each node of SFG and the number of delays $D(\vec{e})$ on every edge are derived from the set of dependence edges $\vec{b}$ at each point in the DG.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} [\vec{b}]$$

3) *I/O mapping.* The SFG node position $\boldsymbol{n}$, and time $t(c)$ of an input of the DG computation $\boldsymbol{c}$ is derived as:

$$\begin{bmatrix} \mathrm{t(c)} \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} [\mathrm{c}]$$

4.2.2 Mapping DG into SFG

Figure 4.2a is the simple DG representation of Figure 4.1. The arrows, the dependence arcs $\vec{e}$, in the same direction are combined into one for the simplicity of the discussion (e.g. the vertical arrow represents inputs nb_i, b_i , and n_i). Each circle represents the processing element, *PE*.

By following the algebraic mapping procedure in Section 4.2.1, two SFGs are derived from DG. SFG-I in Figure 4.2b is the generated SFG by using the projection direction vector $\vec{d} = [0\ 1]$ and default schedule vector $\vec{s}^T = [0\ 1]$, while SFG-II in Figure 4.2c is the generated SFG when the projection vector $\vec{d}$ is [1 0] and schedule vector $\vec{s}^T = [1\ 1]$. The default schedule in the latter SFG is not permissible, thus we chose different vector value to satisfy the two conditions that define the validity of schedule for the given DG. These conditions are:

(1) $\vec{s}^T \vec{d} > 0$,

(2) $\vec{s}^T \vec{e} \geq 0$, for any dependence arc $\vec{e}$.

For the succeeding discussion PE_{DG} represents the processing element of DG, $PE_{SFG\text{-}I}$ represents the processing element of SFG-I and $PE_{SFG\text{-}II}$ represents the processing element of SFG-II .

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

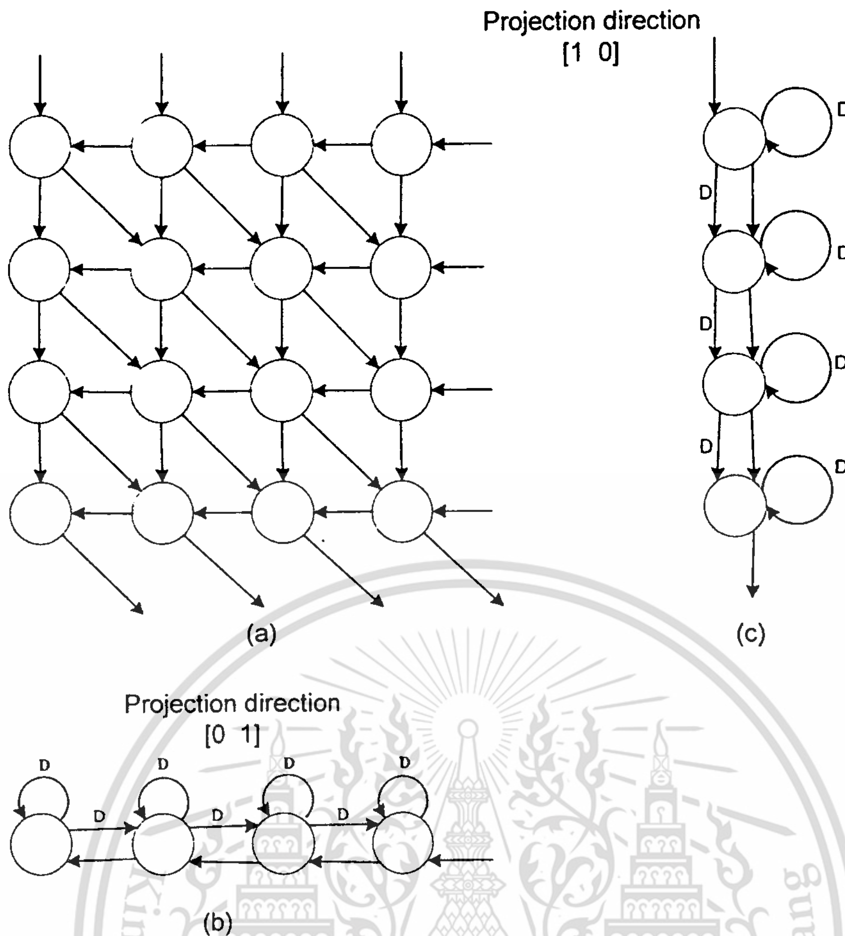


Figure 4.2 (a) Simple Dependence Graph(DG) represention of Systolic Montgomery Algorithm, (b) SFG-I, with a projection vector [0 1], (c) SFG-II, with a projection vector [1 0]

All $PE_{DG}s$ in each column of DG (Figure 4.2a) is projected to each $PE_{SFG\text{-}I}$ (Figure 4.2b). Since each PE_{DG} of every column of DG performs the same operation, thus the architecture of each $PE_{SFG\text{-}I}$ follows the architecture of the $PE_{DG}s$ projected to it. The same thing happens for SFG-II, the $PE_{DG}s$ in each row of DG are projected to each $PE_{SFG\text{-}II}$. But this time, since the rightmost PE_{DG} in each row of DG performs extra operation for the computation of q_i (see Section 4.1), $PE_{SFG\text{-}II}$ should be designed to accommodate this operation.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

4.2.3 Systolic Mapping

Now we are going to systolize the two SFGs in the previous section. It will be shown that if we incorporate pipelining into an SFG array, it will lead to a systolic design. By following the same algebraic procedure in Section 4.2.1, and using a schedule vector $\vec{s}^T = [1\ 2]$, the new fully systolized will be generated, see Figure 4.3.

The two conditions for selecting schedule vector are satisfied;

$$\vec{s}^T\ \vec{d} > 0 \ \text{ and } \ \vec{s}^T\ \vec{e} > 0,$$

It guarantees that every edge of the SFG will have one or more delays ($D(e) \geq 1$), notice from the result of algebraic computation below.

SFG I: Given $\vec{d} = [0\ 1]$, $\vec{s}^T = [1\ 2]$, $P^T = [1\ 0]$

Node mapping:

$$n = P^T\,\mathbf{c} = [1\ 0]\begin{bmatrix} i \\ j \end{bmatrix} = i$$

Arc mapping:

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\vec{b}\,] = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & -1 & 0 \end{bmatrix}$$

Input mapping:

$$\begin{bmatrix} \mathrm{t(c)} \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\mathrm{c}] = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ 1 \end{bmatrix} = \begin{bmatrix} i+2 \\ i \end{bmatrix}$$

Output mapping:

$$\begin{bmatrix} \mathrm{t(c)} \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\mathrm{c}] = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ N \end{bmatrix} = \begin{bmatrix} i+2N \\ i \end{bmatrix}$$

SFG II: Given $\vec{d} = [1\ 0]$, $\vec{s}^T = [1\ 2]$, $P^T = [0\ 1]$

Node mapping:

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

$$n = P^T \mathbf{c} = [0\ 1]\begin{bmatrix} i \\ j \end{bmatrix} = j$$

Arc mapping:

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\vec{b}\,] = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}$$

Input mapping:

$$\begin{bmatrix} \mathrm{t(c)} \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\mathrm{c}] = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ 1 \end{bmatrix} = \begin{bmatrix} i+2 \\ 1 \end{bmatrix}$$

Output:

$$\begin{bmatrix} \mathrm{t(c)} \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix}[\mathrm{c}] = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ N \end{bmatrix} = \begin{bmatrix} i+2N \\ N \end{bmatrix}$$

The systolized SFGs are shown in Figure 4.3. For our implementation, we are going to use the systolized SFG-II in Figure 4.3c for 2 major reasons;

1. *Pipelining period*: the pipelining period of SFG-II is one ($\alpha = 1$), which means that PE utilization is 100%, while pipelining period of SFG-I is two ($\alpha = 2$), PE utilization is reduced to 50%.

Pipelining period for SFG-I:
given $\vec{d} = [0\ 1]$, $\vec{s}^T = [1\ 2]$,
$$\alpha = \vec{s}^T \vec{d} = [1\ 2]\begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0 + 2 = 2$$

Pipelining period for SFG-II:
given $\vec{d} = [1\ 0]$, $\vec{s}^T = [1\ 2]$,
$$\alpha = \vec{s}^T \vec{d} = [1\ 2]\begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0 + 1 = 1$$

2. *Ports*: Based on the computation above, the output ports and the vertical input of SFG-I are located in each node of the array. And in the actual implementation, there are three inputs in a vertical direction, nb_i, b_i, n_i, see Figure 4.1, which means that $3(l+1)$ ports are needed for these three inputs where l is the length of the modulus, plus another $(l+1)$ for the output ports and a single port for operand A,

King Mongkut's Institute of Technology Ladkrabang
This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

with a total of $4(l+1) + 1$ ports. While in SFG-II only one output port is needed located in the last node of the array. Inputs nb_i, b_i, and n_i will be inputted sequentially in the first node and a single input port will be needed at each node for each bit of operand A, this requires a total of $(l+1) + 4$ ports.

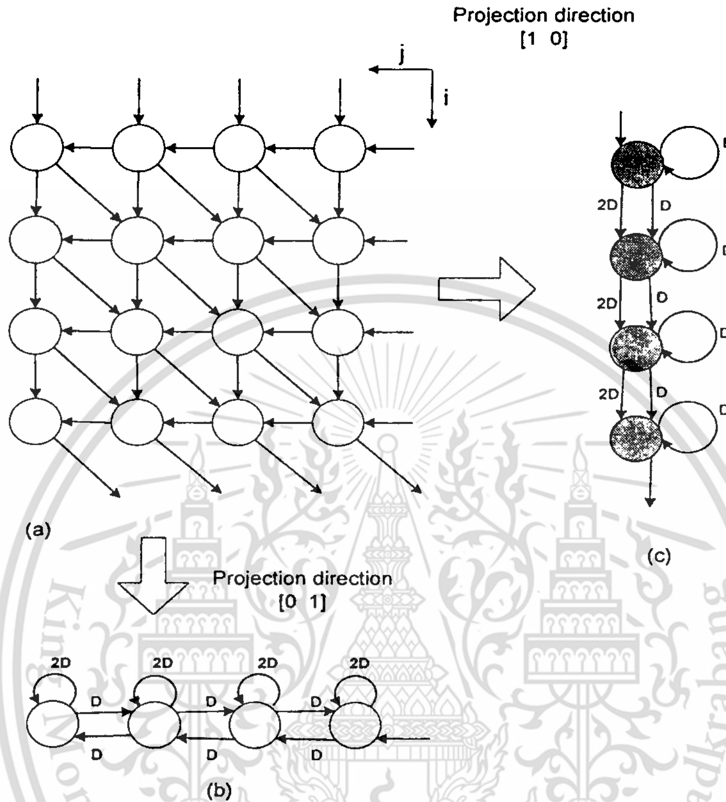


Figure 4.3 (a) Dependence Graph (DG), (b) Systolized SFG-I, and (c) Systolized SFG-II.

The architectural design of each systolized SFG-II node will be discussed in the next section. We named this architecture as SFG-II Multiplier.

4.3 VHDL Design and FPGA Implementation

4.3.1 Architectural Design and Simulation

The VHDL design of the SFG-II Multiplier is in Appendix A, it can be configured to any size of modulus by just changing the value of *generic Len*, *Len* stands for the length of modulus, $Len = l$.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Figure 4.4a shows the top level schematic of our VHDL design. The first node, *cell_0*, receives the inputs, nb_i, b_i, n_i and *Pi* as well as the *select1* and *select2* signals sequentially at every clock cycle, then propagate through the array. The partial product *P* which is initialized to zero (see Algorithm 4.3) will be fed in ports *Pi* and *P0*, LSB at *P0* with two delays and the rest of its bits $p_1, p_2, p_3, \ldots p_l$ will flow through signal *Pi* with one delay each . Each bit of operand *A* is inputted at every node, a_0 in the first node, a_1 in the second node, and so on, where a_l is always zero. *Pout1* and *Pout0* of a certain node are the *Pi* and *P0* input of the node next to it. The last node, *cell_4*, has only one output, *Pout0*, where the final partial product of the Montgomery multiplication process will be outputted sequentially every clock cycle starting with the least significant bit (LSB) to most significant bit (MSB).

The architecture of each node in SFG-II Multiplier is shown in Figure 4.4b. The flip flops FF*i* represent the delays, where *i* is an integer as shown in the figure. Each bit of modulus *N*, operand *B* and precomputed *NB* will be inputted sequentially starting at the least significant bit (LSB) with 2 delays each.

For the simulation and verification, we used Modelsim XE II v5.6a. The funtionality of our design has been verified at *Len* = 4, the simulation result with its corresponding testbench (through *do* macro file) is in Appendix B.

Total execution time (*T*)

From the simulation result in Appendix B, it can be seen that the total execution time of our design is 13 clock cycle, which is equivalent to $3l + 1$ where $l = 4$. Total execution time can be theoretically computed by this equation [13];

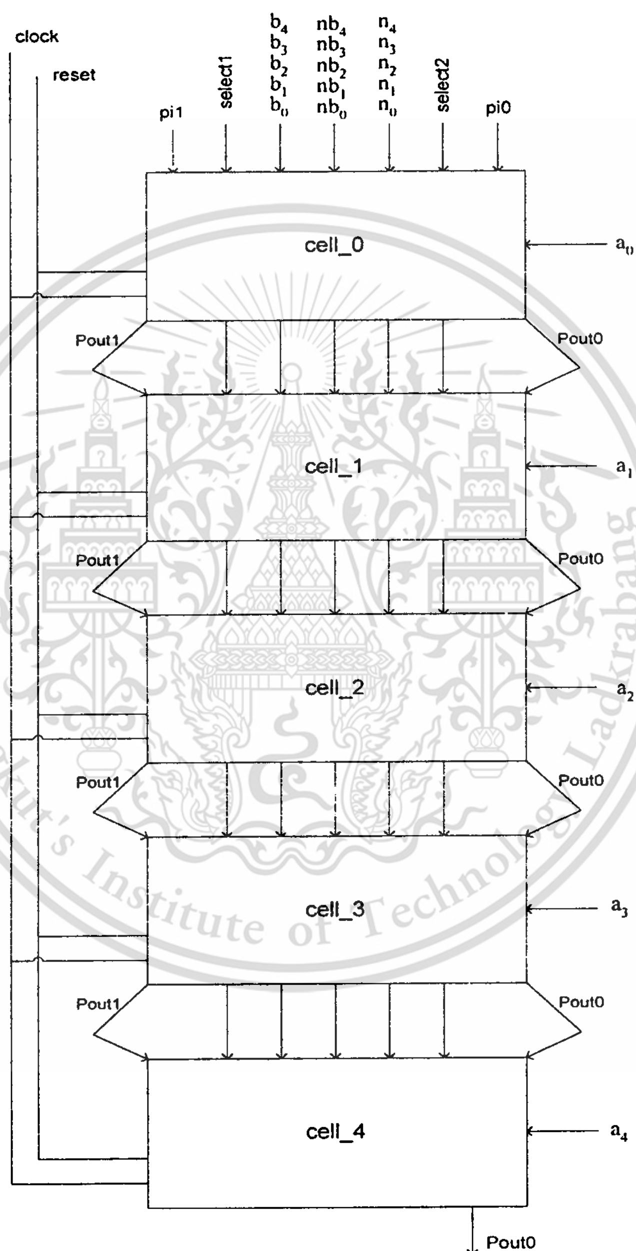
$$T = \max\{\vec{s}^{\,T}(\vec{p} - \vec{q})\} + 1 \qquad \text{where } \vec{p} = [l\ \ l],\ \vec{q} = [0\ \ 0];$$

$$= [1\ \ 2]\begin{bmatrix} l \\ l \end{bmatrix} + 1 \qquad (\vec{p} - \vec{q}) \text{ is the maximum difference of any two points in theDG}$$

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

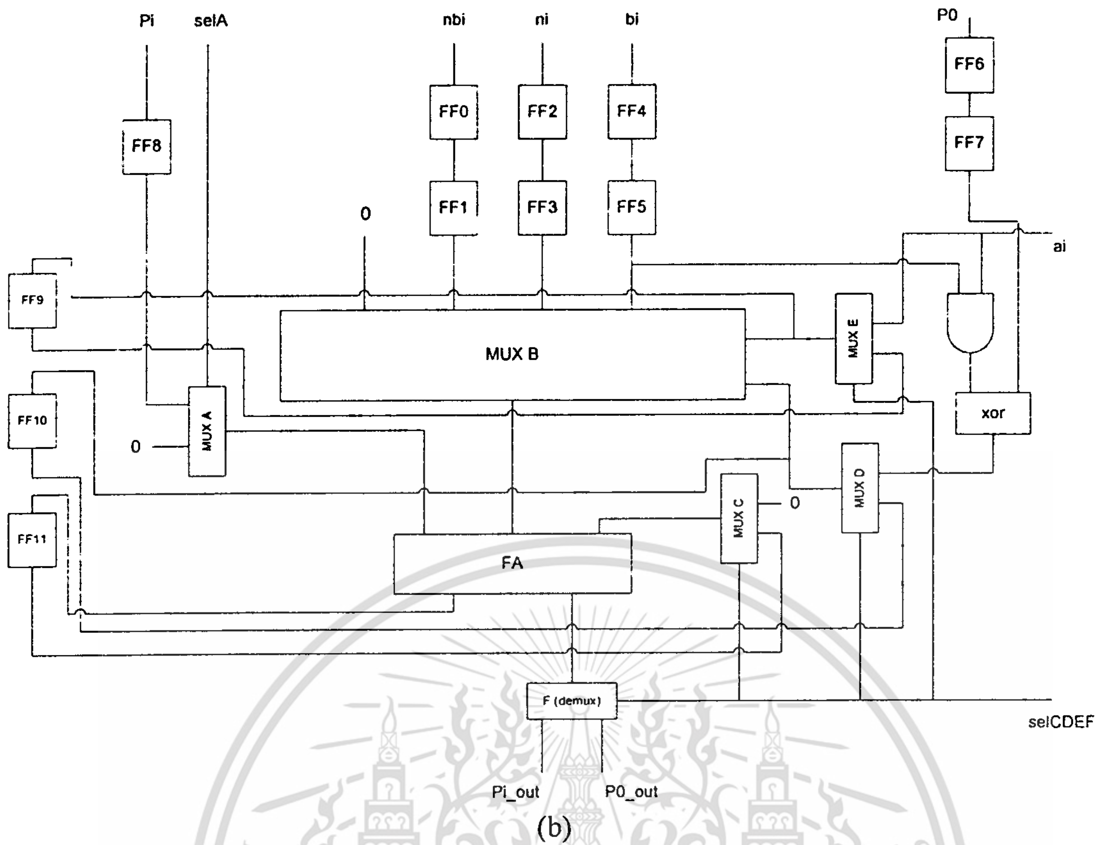
$$= l + 2l + 1$$
$$= 3l + 1$$

It can also be observed that this is equivalent to the ouput mapping derived in our computation in Section 4.2.3, when i is equivalent to the last node $(l + 1)$, the leftmost node of SFG-I or bottom node of SFG-II, in equation $t(c) = i + 2N$.



(a)

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.



(b)

Figure 4.4 (a) Architecture designof Systolized SFG-II when n=4, (b) Typical structure of each processing element in systolized SFG-II.

Latency and Throughput

The latency of our design is $2l+1$ clock cycles, where the LSB of the final partial product is the first output bit. Then the succeeding bits will be outputed sequentially every clock cycle, giving us the throughput of one bit per clock cycle. Table 4.2 shows the performance comparison between [3] and our design in terms of design parameters, though they have almost equivalent amount of figures in this table, the clock cycle time they are using are different which is shown in Table 4.3.

Table 4.2 Performance Comparison: Our design vs. Nedjah's design [3]

Design Parameter	[3]	Ours
Total execution time (ns)	$3l + 3$	$3l + 1$
Latency (ns)	$2l + 3$	$2l + 1$
Throughput	1 bit per clock cycle	1 bit per clock cycle

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

4.3.2 FPGA Implementation Results

The VHDL design in Appendix A has been synthesized and implemented using the Xilinx ISE 5.1i Project Navigator. The implementation device we used is xc2v8000 Virtex2 FPGA. We implemented different length of modulus during the synthesis and implementation. Table 4.3 shows the clock cycle time, the area in terms of CLB count and the time-area product for the different modulo size. We also included results from the implementation done in [3] for comparison. Recall that we adopted the same systolic algorithm from this paper.

As expected, the area of our design is much greater than [3], it is due to the additional multiplexers and flip flops needed for the systolic design. But the clock cycle time of our design is much lower than [3], it is noticeable that the clock cycle time of [3] increase as the size of modulus increases, which is not practical in actual application. While in our design, the clock cycle time is nearly constant, see Figure 4.5 resulting to a much lower area-time product as the size of modulo increases. For modulo size 128 and 256, the area-time products are almost the same, see Figure 4.6. But when the modulo size increases to 512, the area-time product of our design becomes lower than [3]. And the big gap continually increases as the size of modulo increases (e.g. when *Len* = 1024, area-time product is 40% lower than [3]). It can be seen from graph that the area-time product of [3] is exponential while our design is linear. From these results, it shows that our design is a big improvement of [3].

Table 4.3 Performance figures, our design vs. Nedjah's design [3].

Size of Modulus	Area (CLBs)		Clock cycle time (ns)		Area x Time	
	Ours	[3]	Ours	[3]	Ours	[3]
128	1,026	259	5.424	23	5,565	5,957
256	2,180	304	5.835	42	12,720	12,767
512	4,362	492	5.835	76	25,452	37,392
768	6,545	578	5.835	82	38,190	47,396
1,024	8,729	639	5.835	134	50,934	85,626

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

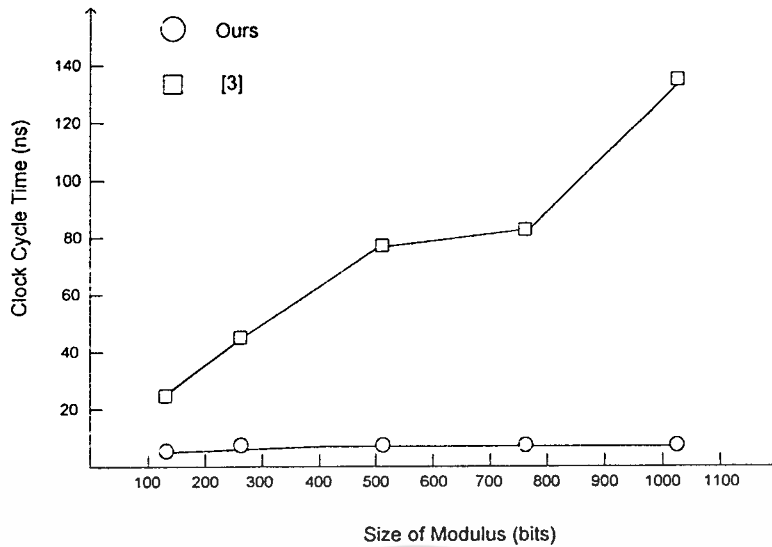


Figure 4.5 Clock cycle time: Nedjah's multiplier vs. Systolized SFG-II multiplier

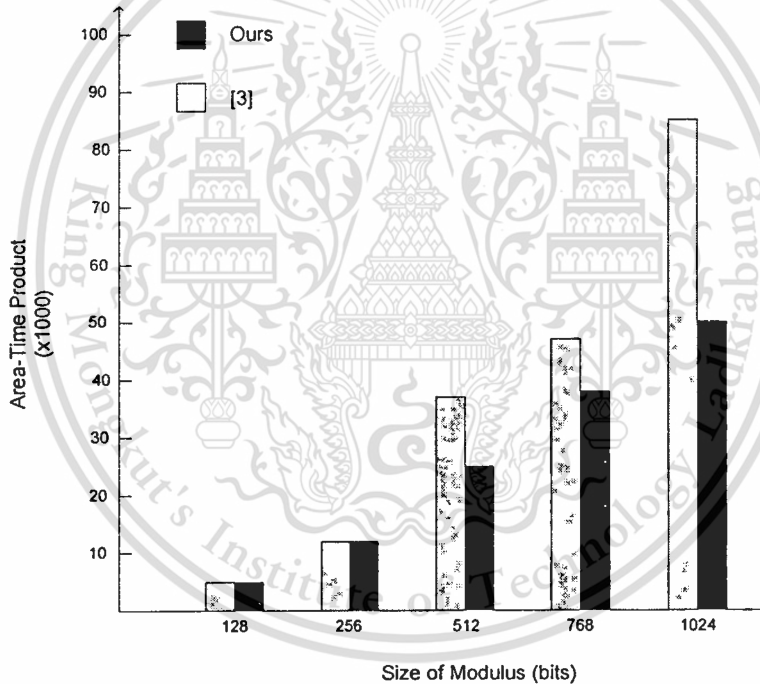


Figure 4.6 Area-Time product: Nedjah's multiplier vs. Systolized SFG-II multiplier

The schematic from synthesis and implementation results is in Appendix C. These results and performance comparison can also be found in Appendix D, the author's paper entitled "An FPGA Implementation of Systolic Array for Montgomery Modular Multiplication".

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Performance Comparison with Other Implementations

We compare our results with other recent implementations, for the design parameters (see Table 4.4), we considered the latency, total execution time (both in clock cycles) and the throughput when l = 1024 in Mb/s, where l is the length of the modulus in bits. Reference [35] is a fully systolic design as ours, [36] is a semi-systolic where the inputs are broadcast while [37] utilized carry-save adder (CSA), [37a] is a five-to-two CSA multiplier and [37b] is four-to-two CSA multiplier.

It can be seen that our design has the highest throughput even if it has greater total execution time in terms of clock cycles. It is due to our resulting clock cycle time that led to a higher throughput. The clock cycle time in nanosecond is shown in Table 4.5 and the comparison can be clearly seen from Figure 4.7. The graph shows that our design, [35] and [36] have nearly constant clock cycle time regardless of the size of the modulus but our design is much lower than the rest, which also means that our design is much faster.

Table 4.4 Design Parameter Comparison

Design	Year	Latency (clock cycles)	Total execution time (clock cycles)	Throughput l = 1024 bits (Mb/s)
Ours	2004	$2l+1$	$3l+1$	171.2
[35]	2003	$2l+1$	$3l+4$	95.6
[36]	2003	-	$2(l+5)$	99.3
[37a]	2003	-	$l+1$	71.0
[37b]	2003	-	$l+2$	76.1

Table 4.5 Performance Comparison: Speed

Modulo Size (l)	Clock cycle time (ns)				
	Ours	[35]	[36]	[37a]	[37b]
512	5.84	10.50	10.07	9.47	9.47
1024	5.84	10.46	11.10	14.08	13.12

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

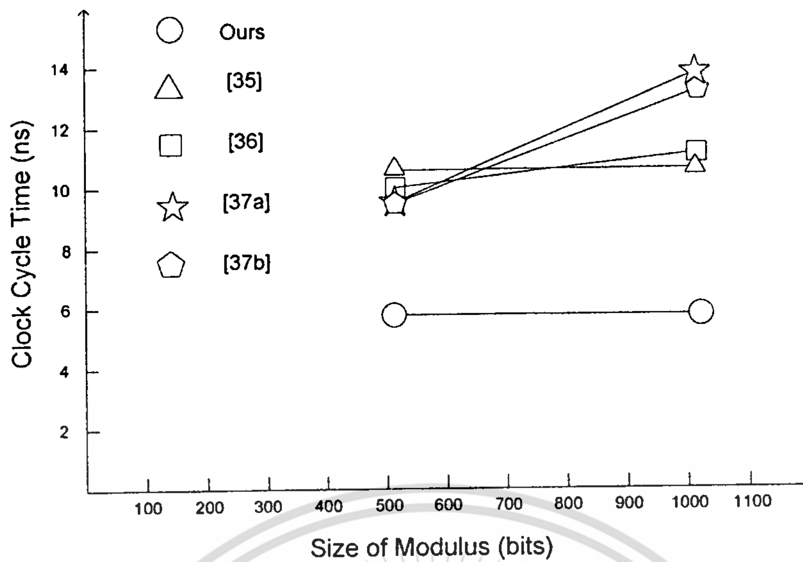


Figure 4.7 Performance Comparison: Speed

Table 4.6 and Figure 4.8 show the area comparison in CLBs. Aside from the area consumed by [36] in CLBs, it also uses the 18x18 built-in multipliers of the FPGA. In our design we need more CLBs which show that the expense of getting a faster speed is additional resources. Considering both area and time, see Table 4.7 and Figure 4.9, area-time product gives us a more reasonable comparison because cost and performance ratio are being considered. The graph shows that though our design consumed the largest area, it is still comparable and competitive especially when l = 1024 with other implementations, we are just a little bit behind [35].

Table 4.6 Performance Comparison: Area

Modulo Size (l)	Area (CLBs)				
	Ours	[35]	[36]	[37a]	[37b]
512	4362	1486	4118*	2481	2845
1024	8729	2853	7167*	5166	5809

* Utilized 18x18-bit multipliers – built-in the FPGA
32 multipliers for 512-bit modulo
62 multipliers for 1024-bit modulo

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

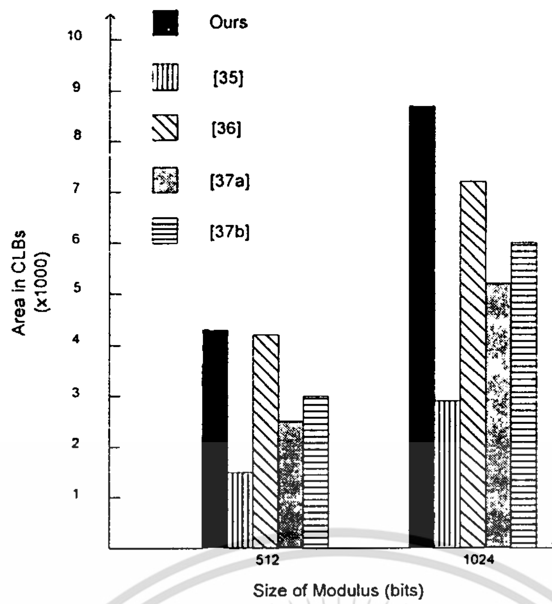


Figure 4.8 Performance Comparison: Area

Table 4.7 Performance Comparison: Area-Time Product

Modulo Size (l)	Area x Time				
	Ours	[35]	[36]	[37a]	[37b]
512	25,474	15,604	41,468	23,495	26,942
1024	50,977	29,837	79,554	72,737	76,214

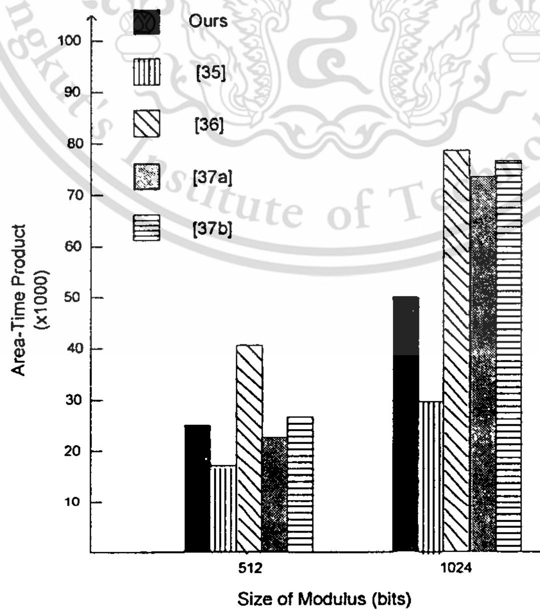


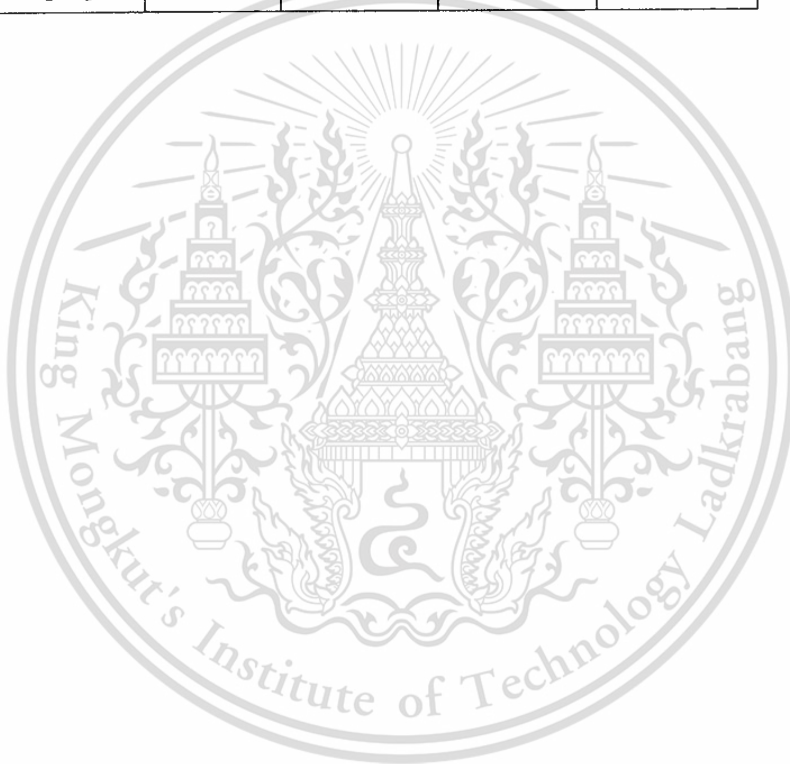
Figure 4.9 Performance Comparison: Area-Time Product

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Table 4.8 shows the actual time needed in one modular multiplication in microseconds. When l = 1024, the execution time is 32.17μs using the multiplier in [35], but if we use SFG-II multiplier (our design), we only need 17.95 μs which is 44% faster than the latter.

Table 4.8 Performance Comparison:
Time for One Modular Multiplication in μs

Design	Size of Modulus (l)			
	128	256	512	1024
Ours	2.09	4.49	8.98	17.95
[35]	3.97	7.69	16.17	32.17



This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Chapter 5

Conclusion and Future Works

5.1 Conclusion

Cryptographic algorithms such as RSA that deals with great number of bits demand for a fast execution of its primitive operation, the modular multiplication. To achieve faster execution we used systolic mapping methodology for our design that will satisfy the requirements in a much higher modulo size. In comparison with reference [3], the results obtained showed that SFG-II Multiplier achieves a much greater and nearly constant frequency at different size of modulus which is very attractive in actual application, because in RSA, the higher the number of bits the more security it has, which needs longer time during exponentiation. This high frequency in turn leads to a higher throughput.

Minimizing IOBs is important for this design because FPGAs has limited numbers of IOBs. In our systolization process, we chose the SFG derived using the projection direction [1 0], yielding a lower number of IOBs and 100% PE utilization compare to the SFG derived using the projection direction [0 1] that requires more IOBs which may further require for more than one FPGAs for its implementation. For our implementation, we utilized FPGA Virtex2 family with enough IOBs for 1024-bit size of modulus that fits our design into only one FPGA.

The architecture design has been coded in VHDL, we used *generic* and *generate* statements for fast and easy reprogramming for different size of modulus, thus it can be used for both RSA and IDEA.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

The SFG-II multiplier is implemented in FPGA which inherent all its flexibility features. This is desirable for this kind of design since security requirement increases with time thus requiring for future developments.

5.2 Future Works

The work described in this research is only the modulo multiplier for RSA algorithm, this multiplier array can be used in the exponentiation operation which is a series of modulo multiplication as discussed in Chapter 2. In Section 2.2.3, it shows that RSA algorithm is a successive squaring and modulo multiplication, thus our modulo multiplier array can be reused for the next multiplication right after the first bit (LSB) of the current multiplication is outputed, that is $2n+1$ clock cycles, this is done by feeding it back to the array. Thus, 100% utilization can be achieved during the entire exponentiation.

Although we derived SFG-I and SFG-II from the dependence graph of the systolic Montgomery modular algorithm, only the systolized SFG-II was implemented. Another design utilizing SFG-I can also be done to show the trade-offs between these two SFGs. Systolized SFG-I will have numerous number of input ports but reduced area due to its simpler processing elements. Techniques on how to increase its 50% PE utilization can also be studied such as pipeline interleaving method.

The synthesis and implementation done in this research is not yet the optimal one. Manual routing for the utilization of specific FPGA's special features where the design will be implemented is suggested to attain higher frequency.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

References

[1] T. Blum, "Modular Exponentiation on Reconfigurable Hardware", *Master Thesis*, Electrical & Computer Engineering Dept., Worcester Polytechnic Institute, Worcester, USA, April 1999.

[2] C.D.Walter, "Systolic modular multiplication", *IEEE Transactions on Computers*, Vol. 42, No. 3, pp. 376-378, March 1993.

[3] N. Nedjah and L. M. Mourelle, "Two hardware implementations for the Montgomery Multiplication: Sequential vs. Parallel", *Proceedings of the 15th. Symposium on Integrated Circuits and Systems Design*, Porto Alegre, RS, Brazil, R. Reis and N. Calazans Eds., IEEE Computer Society Press, pp. 3-8, 2002.

[4] N. Nedjah and L. M. Mourelle, "Three hardware implementations for the Binary Modular Exponentiation: Sequential, Parallel and Systolic", *Proceedings of the 15th. Symposium on Computer Architecture and High Performance Computing*, IEEE Computer Society Press, 2003.

[5] T. Blum and C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", *IEEE Transaction on Computers*, Vol. 50, No. 7, pp.759-764, July 2001.

[6] P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms", *IEEE Transaction on Computers*, Vol. 43, No. 8, pp. 892-898, Aug. 1994.

[7] T. Blum and C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware", *Proc. 14th Symp. Computer Arithmetic*, pp. 70-77, 1999.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

[8] A. Daly and W. Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *FPGA'02*, pp. 40-49, Monterey, CA, February 24-26, 2002.

[9] B. Schneier, *Applied Cryptography, 2nd ed.*, John Wiley & Sons, N. Y., 1996.

[10] C. Kaufman, et. al., *Network Security: Private Communication in Public World*, 1995.

[11] C. K. Koc, "RSA Hardware Implementation", *RSA Labs Technical Report TR-801 v1.0*, RSA Data Security, Inc., CA, August 1995.

[12] A.J. Elbirt and C. Paar, "Towards an FPGA Architecture Optimized for Public-Key Algorithms", *The SPIE's Symposium on Voice, Video, and Communications*, September 1999.

[13] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, Englewood Cliffs, N. J., 1988.

[14] C.-C. Yang, et al, "A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm", *Transaction on Circuit System: Analog and Digital Signal Processing*, Vol. 45, No. 7, July 1998.

[15] Y. J. Jeong and W. P. Burleson, "VLSI Array Algorithms and Architectures for RSA Modular Multiplication", *IEEE Transactions on VLSI Systems*, Vol. 5, No. 2, pp. 211-217, June 1997.

[16] J. Groszschaedl, "The Chinese Remainder Theorem and its Application in a High-Speed RSA Crypto Chip", *Computer Security Applications, 2000, ACSAC '00, 16th Annual Conference*, Dec. 2000.

[17] A. F. Tenca and C. K. Koc, "A Scalable Architecture for Modular Multiplication based on Montgomery's Algorithm", *IEEE Transactions on Computers*, Vol. 52, No. 9, pp.1215-1221, Sept. 2003.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

[18] J. H. Hong and C. W. Wu, "Cellular-Array Modular Multiplier for Fast RSA Public-Key Cryptosystem Based on Modified Booth's Algorithm", *IEEE Transactions on VLSI Systems*, Vol. 11, No. 3, pp. 474-484, June 2003.

[19] W. Stallings, *Cryptography and Network Security: Principles and Practices, 3rd ed.*, Prentice Hall International, N. J., 2003.

[20] D. L. Perry, *VHDL: Programming by Example, 4th ed.*, McGraw Hill Companies, Singapore, 2002.

[21] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals, 2nd ed.*, Prentice Hall, N. J., 2001.

[22] J. R. Armstrong and F. G. Gray, *VHDL Design: Representation and Synthesis, 2nd ed.*, Prentice Hall, N. J., 2000.

[23] K. C. Chang, *Digital Systems Design with VHDL and Synthesis: An Integrated Approach*, IEEE Computer Society, California, 1999.

[24] J. Poldre, "CryptoProcessor PLD001", *Master Thesis*, Department of Computer Science, Tallinn Technical University, June 1998.

[25] C. Kim, "VHDL Implementation of Systolic Modular Multiplications on RSA Cryptosystem", *Master Thesis*, The City College of the City University of New York, Department of Computer Science, N. Y., 2001

[26] K. H. Rosen, *Elementary Number Theory and its Applications, 4th ed.*, Addison Wesley Longman, N. Y., 2000.

[27] I. Koren, *Computer Arithmetic Algorithms, 2nd ed.*, A K Peters, Massachusetts, 2002.

[28] http://support.xilinx.com

[29] A. Menezes, et al., *Handbook of Applied Cryptography*, CRC Press, 1996.

[30] K. Bogart and C. Stein, *Discrete Math in Computer Science*, Dept. of Mathematics and Dept. of Computer Science, Dartmouth College, June 2002.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

[31] J. L. Beuchat, "Modular Multiplication for FPGA Implementation of the IDEA Block Cipher", *Technical Report 2002-32*, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, Lyon Cedex, France, Sept. 2002.

[32] R. Zimmermann, "Efficient VLSI Implementation of Modulo ($2^n \pm 1$) Addition and Multiplication", *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 158-167, Adelaide, Australia, April 1999.

[33] J.-L. Beuchat, "Modular Multiplication for FPGA Implementation of the IDEA Block Cipher", *Proceedings of the Application-Specific Systems, Architectures, and Processors (ASAP'03)*, IEEE Computer Society, 2003.

[34] M. P. Leong, et. al., "A Bit Serial Implementation of the International Data Encryption Algorithm (IDEA)", *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 122-131, April 2000.

[35] S. B. Ors, et. al., *Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array*, Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Computer Society Press, April 2003.

[36] S. H. Tang, et. al., *Modular Exponentiation using Parallel Multipliers*, IEEE International Conference on Field-Programmable Technology (FPT), pp. 52- 59, December 2003.

[37] C. McIvor, et. al., *Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures*, The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, Vol. 1, pp. 379-384, November 2003.

[38] P. Montgomery, *Modular Multiplication Without Trial Division*, Mathematics of Computation, Vol.44, No. 170, pp. 519-521, April 1985.

[39] M. Nazar, S. Kittitornkun and P. Bunyatnoparat, *An FPGA Implementation of Systolic Array for Montgomery Modular Multiplication*, to appear on Ladkrabang Engineering Journal.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Appendix A

VHDL Design of the Systolized SFG-II

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE cells IS

COMPONENT cell0
  port ( clock, resetlatch      : in std_logic;
         a0                     :  in std_logic;
         b, n                   : in std_logic;
         nb                     : in std_logic;
         pi0,pi1                : in std_logic;
         selA, selCDEF          : in std_logic;

         Pout0                  : out std_logic;
         Pout1                  : out std_logic;

         selA_out, selCDEF_out  : out std_logic;
         bb, nn, nbnb           : out std_logic);

END COMPONENT;
---------------------------------------------------------------
COMPONENT basiccell
  port (     clock              : in std_logic;
             resetlatch         : in std_logic;
             a0                 : in std_logic;
             b                  : in std_logic;
             n                  : in std_logic;
             nb                 : in std_logic;

             pi0                : in std_logic;
             pi1                : in std_logic;
             selA               : in std_logic;
             selCDEF            : in std_logic;

             Pout0              : out std_logic;
             Pout1              : out std_logic;

             selA_out, selCDEF_out    : out std_logic;
             bb                       : out std_logic;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                nn                              : out std_logic;
                nbnb                            : out std_logic);

END COMPONENT;
------------------------------------------------------------
COMPONENT lastcell
  port (        clock, resetlatch       : in std_logic;
                a0                      :  in std_logic;
                b, n                    : in std_logic;
                nb                      : in std_logic;

                pi0,pi1                 : in std_logic;
                selA, selCDEF           : in std_logic;
                Pout0                   : out std_logic);

END COMPONENT;

END cells;

------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE parts IS

   COMPONENT mux4to1
        port (  Sel     : in std_logic_vector(0 to 1);
                Din     : in std_logic_vector(0 to 3);
                Y       : out std_logic);
   END COMPONENT;

   COMPONENT mux2to1
        port (  Sel     : in std_logic;
               Din      : in std_logic_vector(0 to 1);
               Y        : out std_logic);
   END COMPONENT;

   COMPONENT FA
        port (  A, B, Cin       : in  std_logic;
                S, Cout         : out std_logic);
   END COMPONENT;

   COMPONENT latch
        port (  CLK, reset, D : in STD_LOGIC;
                Q             : out STD_LOGIC);
   END COMPONENT;

   COMPONENT demux
        port (  Sel           : in std_logic;
                Din           : in std_logic;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                    Dout             : out std_logic_vector(0 to 1));
   END COMPONENT;

END parts;

----------------------------------------------------------------

-- Demux Design

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity demux is
   port (        Sel      : in std_logic;
                 Din      : in std_logic;
                 Dout     : out std_logic_vector(0 to 1));
end demux;

architecture RTL of demux is

begin
  process(sel, Din)
   begin
    if(sel = '0') then
                 Dout(0) <= Din;
                 Dout(1) <= '0';
    else
                 Dout(0) <= '0';
                 Dout(1) <= Din;
    end if;
  end process;
end RTL;

---------------------------------------------------------

-- Latch design

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity latch is
   Port (        CLK, reset, D          : in STD_LOGIC;
                 Q                      : out STD_LOGIC);
end latch;

architecture latch_archi of latch is
   begin

          process(clk, reset)
          begin
                   if (reset = '1') then
```

This material reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                Q <= '0';
              elsif (CLK'event and CLK = '1') then
                Q <= D;
              end if;

     end process;

end latch_archi;
---------------------------------------------------------------

-- MUX 4-1 design

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux4to1 is
   Port (        Sel      : in std_logic_vector(0 to 1);
                 Din      : in std_logic_vector(0 to 3);
                 Y        : out std_logic);
end mux4to1;

architecture RTL3 of mux4to1 is

   begin

     with SEL select
        Y <=   Din(0)  when "00",
               Din(1)  when "01",
               Din(2)  when "10",
               Din(3)  when "11",
         '     X' when others;

end RTL3;
---------------------------------------------------------------

-- MUX 2 to 1 design

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2to1 is
   Port (        Sel      : in std_logic;
                 Din      : in std_logic_vector(0 to 1);
                 Y        : out std_logic);
end mux2to1;

architecture RTL of mux2to1 is

   begin
     with SEL select
           Y <=   Din(0)  when '0',
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                        Din(1)  when '1',
                        'X' when others;
end RTL;
----------------------------------------------------------

--FA design

library IEEE;
use IEEE.std_logic_1164.all;

entity FA is
  port (          A, B, Cin         : in  std_logic;
                  S, Cout           : out std_logic);
end FA;

architecture RTL of FA is
begin
  S      <= A xor B xor Cin;
  COUT   <= (A and B) or (A and Cin) or (B and Cin);
end RTL;

------------------------------------------------------------
------------------------------------------------------------
--------------------------------------
--MAIN PROGRAM-- CELL0--
--------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
------------------------------------------------------------
entity cell0 is

  port (         clock, resetlatch         : in std_logic;
                 a0                        :  in std_logic;
                 b, n                      : in std_logic;
                 nb                        : in std_logic;
                 pi0,pi1                   : in std_logic;
                 selA, selCDEF             : in std_logic;
                 ]
                 Pout0                     : out std_logic;
                 Pout1                     : out std_logic;

                 selA_out, selCDEF_out          : out std_logic;
                 bb, nn, nbnb                   : out std_logic);

end cell0;
-------------------------------------------------------------

architecture RTL of cell0 is

  signal zero             : std_logic;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
  signal y              : std_logic;
  signal q0, q2         : std_logic;      -- in& out of MUX D, 'q2' select signal of MUX B
  signal a2             : std_logic;      -- in & out of MUX E, 'a2' select signal of MUX B

  signal carry_in, q1, a1       : std_logic;
  signal v                      : std_logic;      -- in & out of MUX C, carry_in(0) = '0'
  signal u                      : std_logic;      -- in & out of MUX A, 'pi' starts at pi(1)
  signal x                      : std_logic;     -- output of MUX B
  signal carry_out, pi_next  : std_logic;       -- output of FA

-----------------------------------------------------------

-- instantiation

 COMPONENT mux4to1
      port(   Sel      : in std_logic_vector(0 to 1);
              Din      : in std_logic_vector(0 to 3);
      Y    : out std_logic);
 END COMPONENT;

 COMPONENT mux2to1
      port (  Sel      : in std_logic;
              Din      : in std_logic_vector(0 to 1);
              Y        : out std_logic);
 END COMPONENT;

 COMPONENT FA
      port ( A, B, Cin         : in  std_logic;
              S, Cout          : out std_logic);
 END COMPONENT;

 COMPONENT latch
  Port (        CLK, reset, D : in STD_LOGIC;
                 Q             : out STD_LOGIC);
 END COMPONENT;

 COMPONENT demux
      port (  Sel      : in std_logic;
              Din      : in std_logic;
              Dout     : out std_logic_vector(0 to 1));
 END COMPONENT;

-- end of instantiation
-----------------------------------------------------------

begin

 zero <=  '0';
 -------------------------------------
  y  <=  a0 and b after 0 ns;
 q0  <=  y xor pi0 after 0ns;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
--------------------------------------

        latchA:  latch port map (
                 clk   => clock,
                 reset => resetlatch,
                 D     => a2,
                 Q     => a1);

        latchQ:  latch port map (
                 clk   => clock,
                 reset => resetlatch,
                 D     => q2,
                 Q     => q1);

        latchCARRY:  latch port map (
                 clk   => clock,
                 reset => resetlatch,
                 D     => carry_out,
                 Q     => carry_in);

--------------------------------------------------------------

        muxC:  mux2to1 port map (
            sel     => selCDEF,
            Din(0)  => zero,
            Din(1)  => carry_in,
              Y     => v);

        muxD:  mux2to1 port map (
            sel     => selCDEF,
            Din(0)  => q0,          -- out of XOR gate at the first stage
            Din(1)  => q1,          -- out of latch at the succeeding stages
              Y     => q2);         -- one of the select signal of MUXB

        muxE:  mux2to1 port map (
            sel     => selCDEF,
            Din(0)  => a0,
            Din(1)  => a1,
              Y     => a2);

--------------------------------------------------------------

        muxA:  mux2to1 port map (
            sel     => selA,
            Din(0)  => zero,
            Din(1)  => pi1,
              Y     => u);

--------------------------------------------------------------

        muxB:  mux4to1 port map (
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                    sel(0)  => q2,
                    sel(1)  => a2,
                    Din(0) => zero,
                    Din(1) => b,
                    Din(2) => n,
                    Din(3) => nb,
                      Y     => x);

------------------------------------------------------------

                FA1:  FA port map (
                     A        => u,
                    B         => x,
                    Cin       => v,
                    S         => pi_next,
                    Cout      => carry_out);

------------------------------------------------------------

                demuxF:  demux port map (
                    sel        => selCDEF,
                    Din        => pi_next,
                    Dout(0) => Pout0,
                    Dout(1) => Pout1);

   selA_out <= selA;
   selCDEF_out <= selCDEF;

   bb <= b;
   nn <= n;
   nbnb <= nb;

end RTL;

----------------------------
--END OF CELL0--
----------------------------
------------------------------------------------------------
----------------------------------------------
--MAIN PROGRAM OF BASICCELL--
----------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;

entity basiccell is

  port (        clock, resetlatch         : in std_logic;
                a0                        :  in std_logic;
                b, n                      : in std_logic;
                nb                        : in std_logic;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
            pi0,pi1               : in std_logic;
            selA, selCDEF         : in std_logic;

            Pout0                 : out std_logic;
            Pout1                 : out std_logic;

            selA_out, selCDEF_out        : out std_logic;
            bb, nn, nbnb                 : out std_logic);

end basiccell;
---------------------------------------------------------------
architecture RTL of basiccell is

  signal zero                                   : std_logic;
  signal selA_out1, selA_out2                   :  std_logic;
  signal selCDEF_out1, selCDEF_out2             :  std_logic;

  signal bout1, bout2              : std_logic;
  signal nout1, nout2              : std_logic;
  signal nbout1, nbout2            : std_logic;

  signal pi_out0, pi_out1          : std_logic;
  signal p                         : std_logic;     -- for added 1 delay at p0
  signal y                         : std_logic;

  signal q0, q2                    : std_logic;     -- in & out of MUX D
  signal a2                        : std_logic;     -- in & out of MUX E
  signal carry_in, q1, a1          :  std_logic;
  signal v                         : std_logic;     -- in & out of MUX C
  signal u                         : std_logic;     -- in & out of MUX A
  signal x                         : std_logic;     -- output of MUX B

  signal carry_out, pi_next        : std_logic;     -- output of FA
  signal pii0, pii1                : std_logic;

---------------------------------------------------------------
-- instantiation

 COMPONENT mux4to1
       port(  Sel     : in std_logic_vector(0 to 1);
              Din     : in std_logic_vector(0 to 3);
              Y       : out std_logic);
 END COMPONENT;

 COMPONENT mux2to1
       port (  Sel     : in std_logic;
               Din     : in std_logic_vector(0 to 1);
                Y       : out std_logic);
 END COMPONENT;
```

This document is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
  COMPONENT FA
        port (  A, B, Cin        : in  std_logic;
                S, Cout          : out std_logic);
  END COMPONENT;

  COMPONENT latch
   Port (       CLK, reset, D : in STD_LOGIC;
                Q              : out STD_LOGIC);
  END COMPONENT;

  COMPONENT demux
        port (  Sel     : in std_logic;
                Din     : in std_logic;
                Dout    : out std_logic_vector(0 to 1));
  END COMPONENT;

-- end of instantiation
---------------------------------------------------------------------
begin

  zero <=  '0';
  ---------------------------------------------------

                latchselA1:  latch port map (
                        clk   => clock,
                        reset => resetlatch,
                        D     => selA,
                        Q     => selA_out1);

                latchselA2:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    => selA_out1,
                        Q    => selA_out2);

                latchselselCDEF1:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    => selCDEF,
                        Q    => selCDEF_out1);

                latchselCDEF2:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    =>selCDEF_out1,
                        Q    => selCDEF_out2);


                latchB1:  latch port map (
                        clk => clock,
                        reset => resetlatch,
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
            D    => b,
            Q    => bout1);

latchB2:  latch port map (
            clk  => clock,
            reset => resetlatch,
            D    => bout1,
            Q    => bout2);

latchN1:  latch port map (
            clk  => clock,
            reset => resetlatch,
            D    => n,
            Q    => nout1);

latchN2:  latch port map (
            clk  => clock,
            reset => resetlatch,
            D    => nout1,
            Q    => nout2);

latchNB1:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => nb,
            Q   => nbout1);

latchNB2:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => nbout1,
            Q   => nbout2);
------------------------------------------------------------

latchP0_1:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => pi0,
            Q   => p);

latchP0_2:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => p,
            Q   => pi_out0);

latchP1_1:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => pi1,
            Q   => pi_out1);
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
------------------------------------------------------------
 y  <= a0 and bout2 after 0 ns;
q0 <= y xor pi_out0 after 0 ns;
------------------------------------------------------------

   muxC:  mux2to1 port map (
                    sel       => selCDEF_out2,
                    Din(0)  => zero,
                    Din(1)  => carry_in,
                       Y       => v);

              muxD:  mux2to1 port map (
                    sel       => selCDEF_out2,
                    Din(0)  => q0,            -- out of XOR gate at the first stage
                    Din(1)  => q1,            -- out of latch at the succeeding stages
                       Y       => q2);          -- one of the select signals of MUXB

              muxE:  mux2to1 port map (
                    sel       => selCDEF_out2,
                    Din(0)  => a0,
                    Din(1)  => a1,
                       Y       => a2);
------------------------------------------------------------

              latchA:  latch port map (
                         clk => clock,
                         reset => resetlatch,
                         D   => a2,
                         Q   => a1);

              latchQ:  latch port map (
                         clk => clock,
                         reset => resetlatch,
                         D   => q2,
                         Q   => q1);

              latchCARRY:  latch port map (
                         clk => clock,
                         reset => resetlatch,
                         D   => carry_out,
                         Q   => carry_in);

------------------------------------------------------------

              muxA:  mux2to1 port map (
                    sel       => selA_out2,
                    Din(0)  => zero,
                    Din(1)  => pi_out1,
                       Y       => u);
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
------------------------------------------------------------

                muxB:  mux4to1 port map (
                       sel(0)   => q2,
                       sel(1)   => a2,
                       Din(0) => zero,
                       Din(1) => bout2,
                       Din(2) => nout2,
                       Din(3) => nbout2,
                         Y      => x);

------------------------------------------------------------

                FA1:  FA port map (
                       A    => u,
                       B    => x,
                       Cin => v,
                       S    => pi_next,
                       Cout => carry_out);

------------------------------------------------------------
                demuxF:  demux port map (
                       sel        => selCDEF_out2,
                       Din        => pi_next,
                       Dout(0) => pii0,
                       Dout(1) => pii1);

   Pout0 <=  pii0;
   Pout1 <=  pii1;

   selA_out <= selA_out2;
   selCDEF_out <= selCDEF_out2;

   bb <= bout2;
   nn <= nout2;
   nbnb <= nbout2;

end RTL;

--------------------------------
-- END OF BASICCELL --
--------------------------------

------------------------------------------------------------
----------------------------------------------
--MAIN PROGRAM OF LAST CELL--
----------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
```

This ... used for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
entity lastcell is

  port (        clock, resetlatch         : in std_logic;
                a0                        :  in std_logic;
                b, n                      : in std_logic;
                nb                        : in std_logic;

                pi0,pi1                   : in std_logic;
                selA, selCDEF             : in std_logic;

                Pout0                     : out std_logic);

end lastcell;
--------------------------------------------------------------
architecture RTL of lastcell is

   signal p                      : std_logic;  -- for added 1 delay at p0
   signal zero                   : std_logic;

   signal selA_out1, selA_out2              : std_logic;
   signal selCDEF_out1, selCDEF_out2        : std_logic;

   signal bout1, bout2           : std_logic;
   signal nout1, nout2           : std_logic;
   signal nbout1, nbout2         : std_logic;

   signal pi_out0, pi_out1       : std_logic;
   signal y                      : std_logic;

   signal q0, q2                 : std_logic;     -- in & out of MUX D
   signal a2                     : std_logic;     -- in & out of MUX E
   signal carry_in, q1, a1       : std_logic;
   signal v                      : std_logic;     -- in & out of MUX C
   signal u                      : std_logic;     -- in & out of MUX A
   signal x                      : std_logic;     -- output of MUX B

   signal carry_out, pi_next     : std_logic;     -- output of FA
------------------------------------------------------
-- instantiation

  COMPONENT mux4to1
        port(  Sel     : in std_logic_vector(0 to 1);
               Din     : in std_logic_vector(0 to 3);
               Y       : out std_logic);
  END COMPONENT;

  COMPONENT mux2to1
        port (  Sel    : in std_logic;
                Din    : in std_logic_vector(0 to 1);
                Y      : out std_logic);
  END COMPONENT;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
  COMPONENT FA
        port (  A, B, Cin        : in  std_logic;
                S, Cout          : out std_logic);
  END COMPONENT;

  COMPONENT latch
        port (  CLK, reset, D : in STD_LOGIC;
                Q                : out STD_LOGIC);
  END COMPONENT;

-- end of instantiation
----------------------------------------------------------------

begin

  zero <=  '0';
  ------------------------------------------------

                latchselA1:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D     => selA,
                        Q    => selA_out1);

                latchselA2:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    => selA_out1,
                        Q    => selA_out2);

                latchselselCDEF1:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    => selCDEF,
                        Q    => selCDEF_out1);

                latchselCDEF2:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D    =>selCDEF_out1,
                        Q    => selCDEF_out2);


                latchB1:  latch port map (
                        clk  => clock,
                        reset => resetlatch,
                        D     => b,
                        Q     => bout1);

                latchB2:  latch port map (
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.
King Mongkut's Institute of Technology Ladkrabang

```
clk  => clock,
          reset => resetlatch,
          D    => bout1,
          Q    => bout2);

latchN1:  latch port map (
          clk  => clock,
          reset => resetlatch,
          D    => n,
          Q    => nout1);

latchN2:  latch port map (
          clk  => clock,
          reset => resetlatch,
          D    => nout1,
          Q    => nout2);

latchNB1:  latch port map (
           clk => clock,
           reset => resetlatch,
           D   => nb,
           Q   => nbout1);

latchNB2:  latch port map (
           clk => clock,
           reset => resetlatch,
           D   => nbout1,
           Q   => nbout2);
------------------------------------------------------------------

latchP0_1:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => pi0,
            Q   => p);

latchP0_2:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => p,
            Q   => pi_out0);

latchP1_1:  latch port map (
            clk => clock,
            reset => resetlatch,
            D   => pi1,
            Q   => pi_out1);

------------------------------------------------------------------
y <= a0 and bout2 after 0 ns;
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
q0 <= y xor pi_out0 after 0 ns;
------------------------------------------------------------
                latchA:  latch port map (
                        clk => clock,
                        reset => resetlatch,
                        D   => a2,
                        Q   => a1);

                latchQ:  latch port map (
                        clk => clock,
                        reset => resetlatch,
                        D   => q2,
                        Q   => q1);

                latchCARRY:  latch port map (
                        clk => clock,
                        reset => resetlatch,
                        D   => carry_out,
                        Q   => carry_in);

------------------------------------------------------------

                muxC:  mux2to1 port map (
                    sel      => selCDEF_out2,
                    Din(0)  => zero,
                    Din(1)  => carry_in,
                       Y      => v);

                muxD:  mux2to1 port map (
                    sel      => selCDEF_out2,
                    Din(0)  => q0,
                    Din(1)  => q1,
                       Y      => q2);

                muxE:  mux2to1 port map (
                    sel      => selCDEF_out2,
                    Din(0)  => a0,
                    Din(1)  => a1,
                       Y      => a2);

------------------------------------------------------------
                muxA:  mux2to1 port map (
                    sel      => selA_out2,
                    Din(0)  => zero,
                    Din(1)  => pi_out1,
                       Y      => u);

------------------------------------------------------------

                muxB:  mux4to1 port map (
                    sel(0)  => q2,
```

This material is for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
                    sel(1)  => a2,
                    Din(0) => zero,
                    Din(1) => bout2,
                    Din(2) => nout2,
                    Din(3) => nbout2,
                      Y     => x);

------------------------------------------------------------
               FA1:  FA port map (
                    A    => u,
                    B    => x,
                    Cin  => v,
                    S    => pi_next,
                    Cout => carry_out);

------------------------------------------------------------

   Pout0 <=  pi_next;


end RTL;

------------------------------
-- END OF LASTCELL --
------------------------------
------------------------------------------------------------
------------------------------------------------------
--MAIN PROGRAM--CASCADED CELLS--
------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;

entity cascadecell is
  Generic ( Len: in integer :=4);

  port (        clk, rst        : in std_logic;
                a_in            : in std_logic_vector(Len downto 0);
                b_in, n_in      : in std_logic;
                nb_in           : in std_logic;

                Ri0, Ri1        : in std_logic;
                select1         : in std_logic;
                select2         : in std_logic;
                Rout0           : out std_logic);

end cascadecell;
------------------------------------------------------------
architecture RTL of cascadecell is
------------------------------------------------------------
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

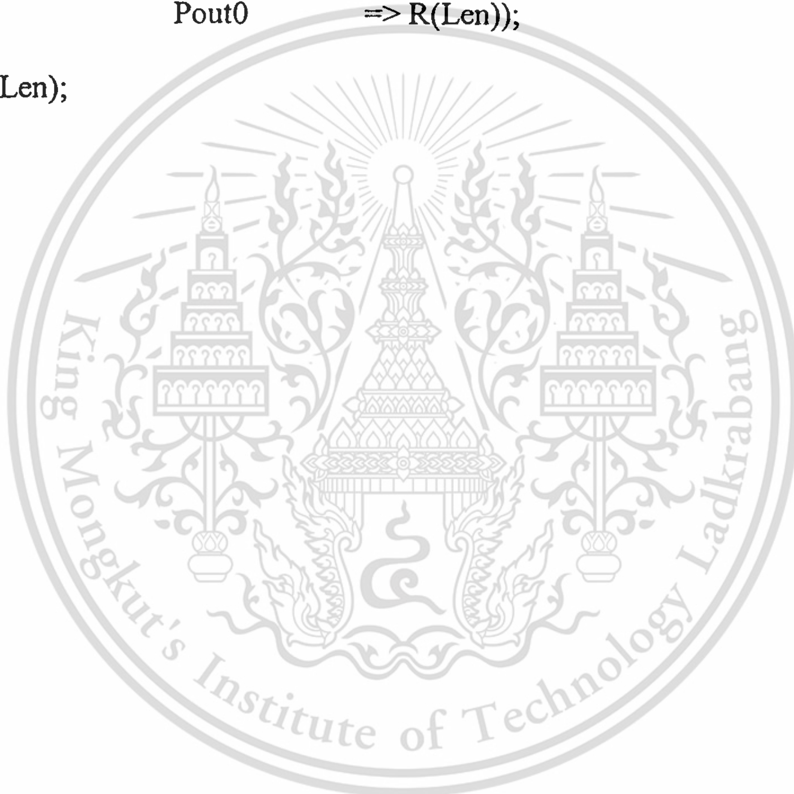
```
COMPONENT cell0
  port (        clock, resetlatch        : in std_logic;
                a0                       :  in std_logic;
                b, n                     : in std_logic;
                nb                       : in std_logic;

                pi0,pi1                  : in std_logic;
                selA, selCDEF            : in std_logic;

                Pout0           : out std_logic;
                Pout1           : out std_logic;

                selA_out, selCDEF_out        : out std_logic;
                bb, nn, nbnb                 : out std_logic);

END COMPONENT;
-----------------------------------------------------------------
COMPONENT basiccell
  port (        clock           : in std_logic;
                resetlatch      : in std_logic;
                a0              :  in std_logic;
                b               : in std_logic;
                n               : in std_logic;
                nb              : in std_logic;

                pi0             : in std_logic;
                pi1             : in std_logic;
                selA            : in std_logic;
                selCDEF         : in std_logic;

                Pout0           : out std_logic;
                Pout1           : out std_logic;

                selA_out, selCDEF_out        : out std_logic;
                bb                           : out std_logic;
                nn                           : out std_logic;
                nbnb                         : out std_logic);

END COMPONENT;
-----------------------------------------------------------------
COMPONENT lastcell

  port (        clock, resetlatch        : in std_logic;
                a0                       :  in std_logic;
                b, n                     : in std_logic;
                nb                       : in std_logic;

                pi0,pi1                  : in std_logic;
                selA, selCDEF            : in std_logic;

                Pout0                    : out std_logic);
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
END COMPONENT;
---------------------------------------------------------------
   signal bsig, nsig, nbsig          : std_logic_vector(0 to Len-1);
   signal S1, S2                     : std_logic_vector(0 to Len-1);
   signal R                          : std_logic_vector(0 to Len);
   signal RR                         : std_logic_vector(0 to Len-1);

---------------------------------------------------------------
 begin
---------------------------------------------------------------

        first_cell:     cell0 port map (
                        clock           => clk,
                        resetlatch      => rst,
                        a0              => a_in(0),
                        b               => b_in,
                        n               => n_in,
                        nb              => nb_in,
                        pi0             => Ri0,
                        pi1             => Ri1,
                        selA            => select1,
                        selCDEF         => select2,
                        Pout0           => R(0),
                        Pout1           => RR(0),
                        selA_out        => S1(0),
                        selCDEF_out     => S2(0),
                        bb              => bsig(0),
                        nn              => nsig(0),
                        nbnb            => nbsig(0));

    cascade:    for k in 1 to (Len-1) generate

                cell_next:  basiccell port map (
                        clock           => clk,
                        resetlatch      => rst,
                        a0              => a_in(k),
                        b               => bsig(k-1),
                        n               => nsig(k-1),
                        nb              => nbsig(k-1),
                        pi0             => R(k-1),
                        pi1             => RR(k-1),
                        selA            => S1(k-1),
                        selCDEF         => S2(k-1),
                        Pout0           => R(k),
                        Pout1           => RR(k),
                        selA_out        => S1(k),
                        selCDEF_out     => S2(k),
                        bb              => bsig(k),
                        nn              => nsig(k),
                        nbnb            => nbsig(k));
```

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
        end generate;

    last_cell:      lastcell port map (
                        clock       => clk,
                        resetlatch  => rst,
                        a0          => a_in(Len),
                        b           => bsig(Len-1),
                        n           => nsig(Len-1),
                        nb          => nbsig(Len-1),
                        pi0         => R(Len-1),
                        pi1         => RR(Len-1),
                        selA        => S1(Len-1),
                        selCDEF     => S2(Len-1),
                        Pout0       => R(Len));

  Rout0 <=  R(Len);

end RTL;
```



This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Appendix B

Testbench and Simulation Results

Testbench1

```
force clk 1 0, 0 50 -repeat 100
force rst 1 0, 0 110

force a_in 01011
force Ri0 0
force Ri1 0

force b_in 1
force n_in 1
force nb_in 0
force select1 1
force select2 0

run 200

force select2 1
force b_in 1
force n_in 1
force nb_in 1

run 100

force b_in 0
force n_in 1
force nb_in 0

run 100

force b_in 1
force n_in 1
force nb_in 1

run 100

force b_in 0
force n_in 0
force nb_in 1
force select1 0

run 100

force b_in 0
force n_in 0
force nb_in 0
force select1 0
force select2 0

run 800
```

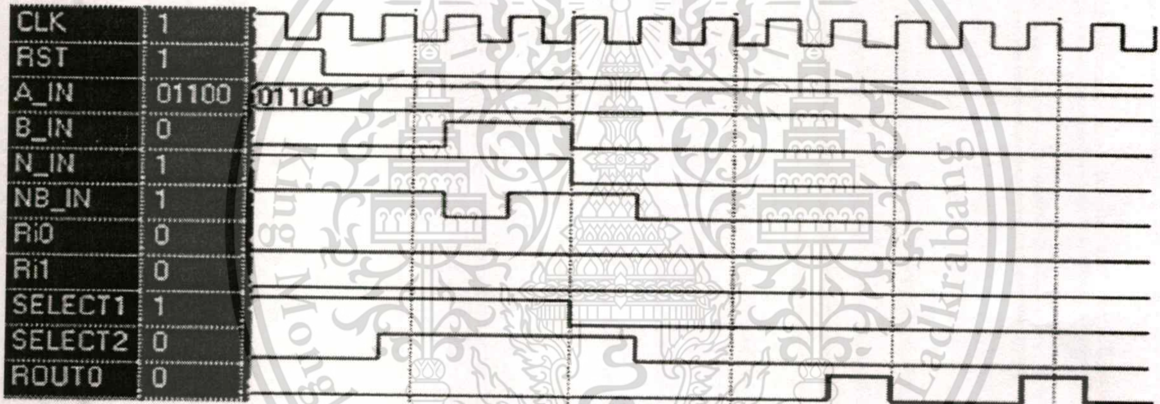
Testbench2

```
force clk 1 0, 0 50 -repeat 100
force rst 1 0, 0 110

force a_in 01100
force Ri0 0
force Ri1 0

force b_in 0
force n_in 1
force nb_in 1
force select1 1
force select2 0

run 200

force select2 1
force b_in 0
force n_in 1
force nb_in 1

run 100
```

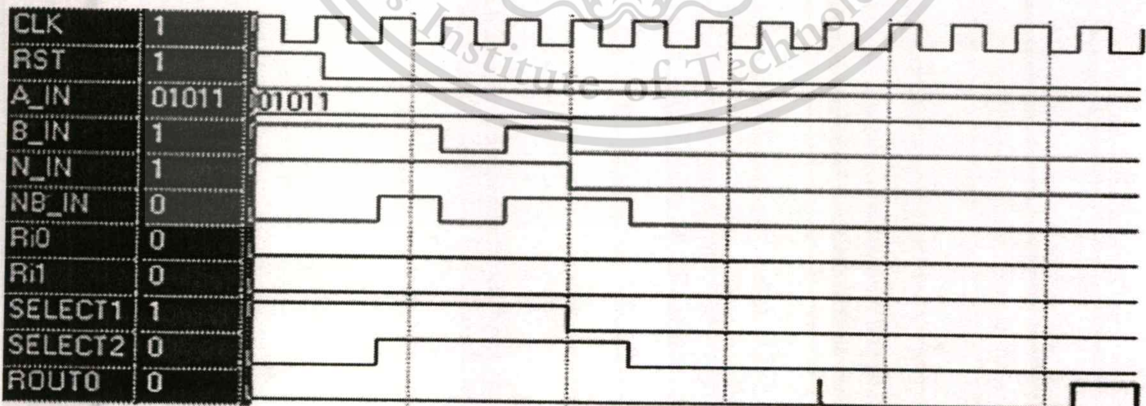
This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

```
force b_in 1
force n_in 1
force nb_in 0

run 100

force b_in 1
force n_in 1
force nb_in 1

run 100

force b_in 0
force n_in 0
force nb_in 1
force select1 0

run 100

force b_in 0
force n_in 0
force nb_in 0
force select1 0
force select2 0

run 800
```

Simulation results



(a)



(b)

Figure B.1 (a) Output waveform of first simulation (Testbench1)

(b) Output waveform of second simulation (Testbench2)

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

Appendix C

Synthesis Results

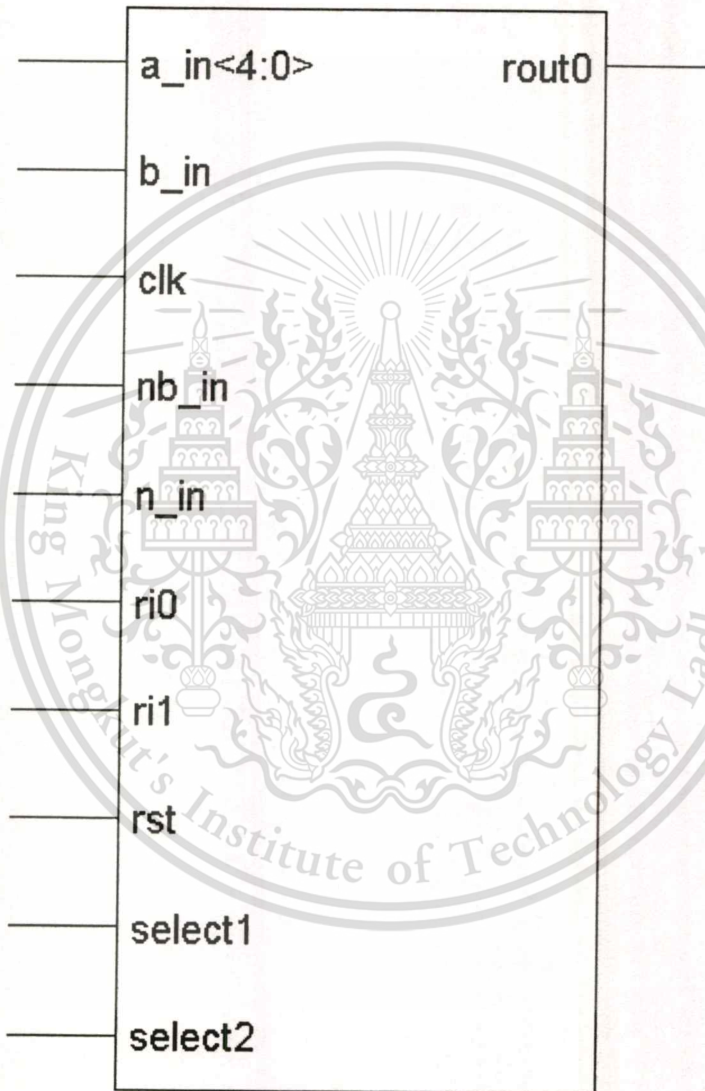


Figure C.1 Top level Schematic

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

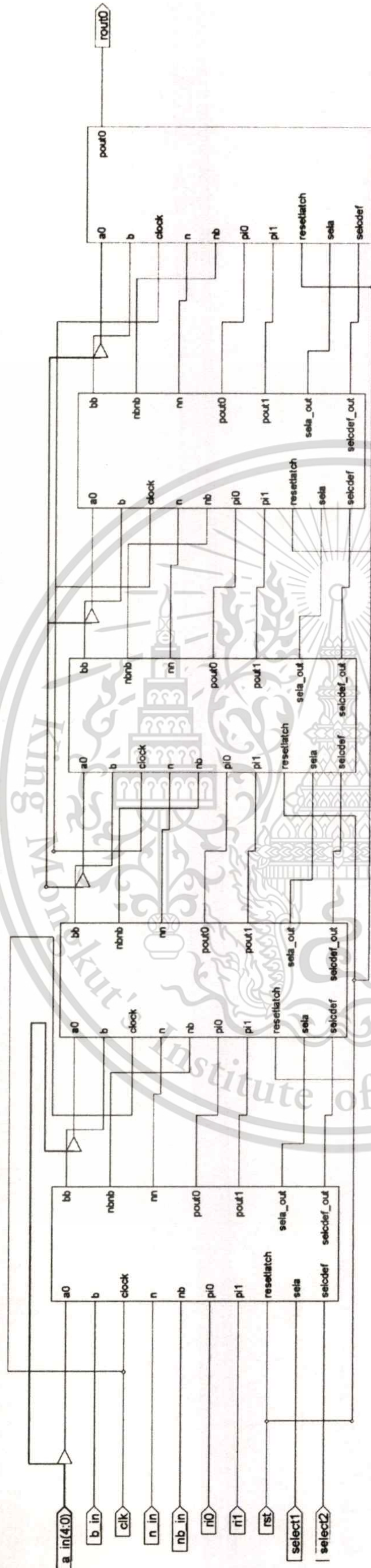


Figure C.2 Cascaded Cells

King Mongkut's Institute of Technology Ladkrabang

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

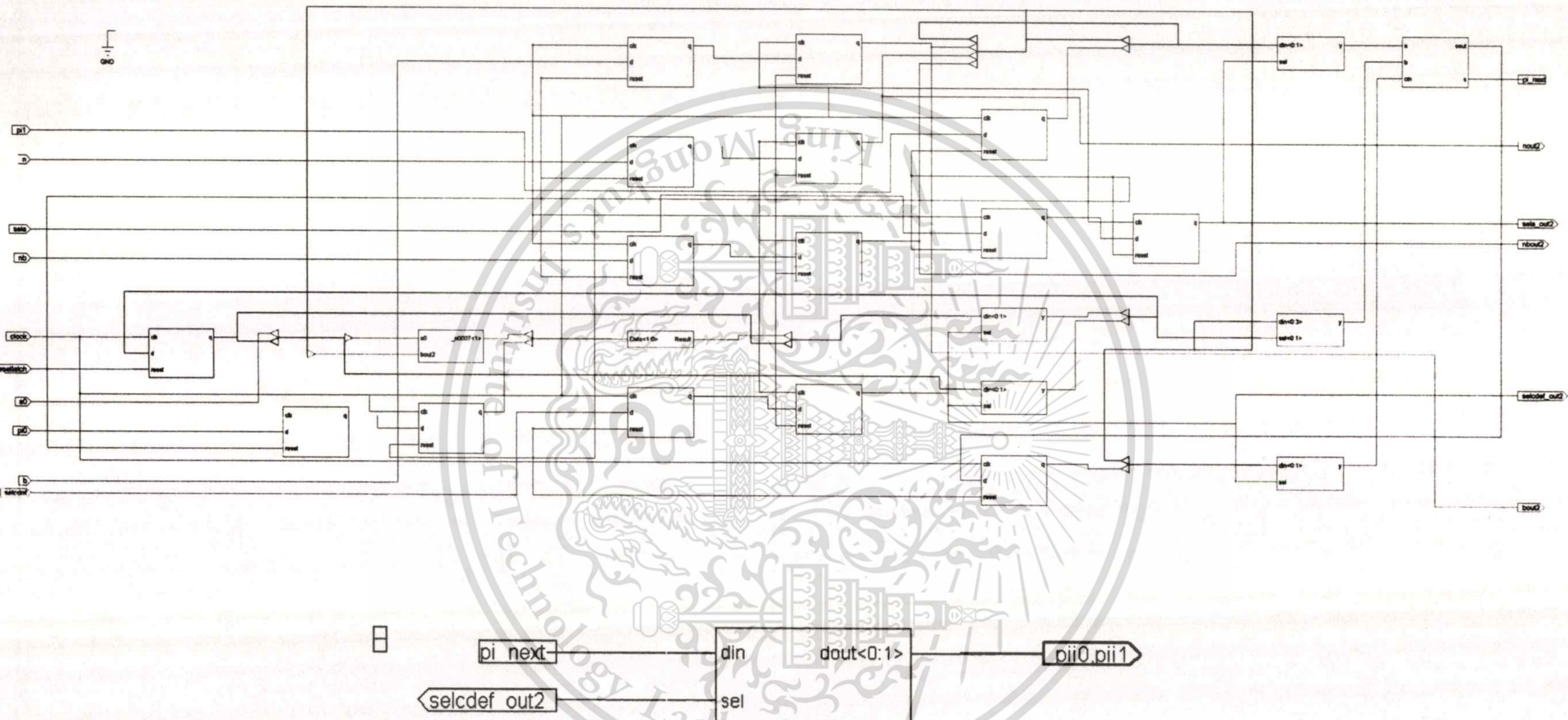


Figure C.3 Basic Cell Schematic

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

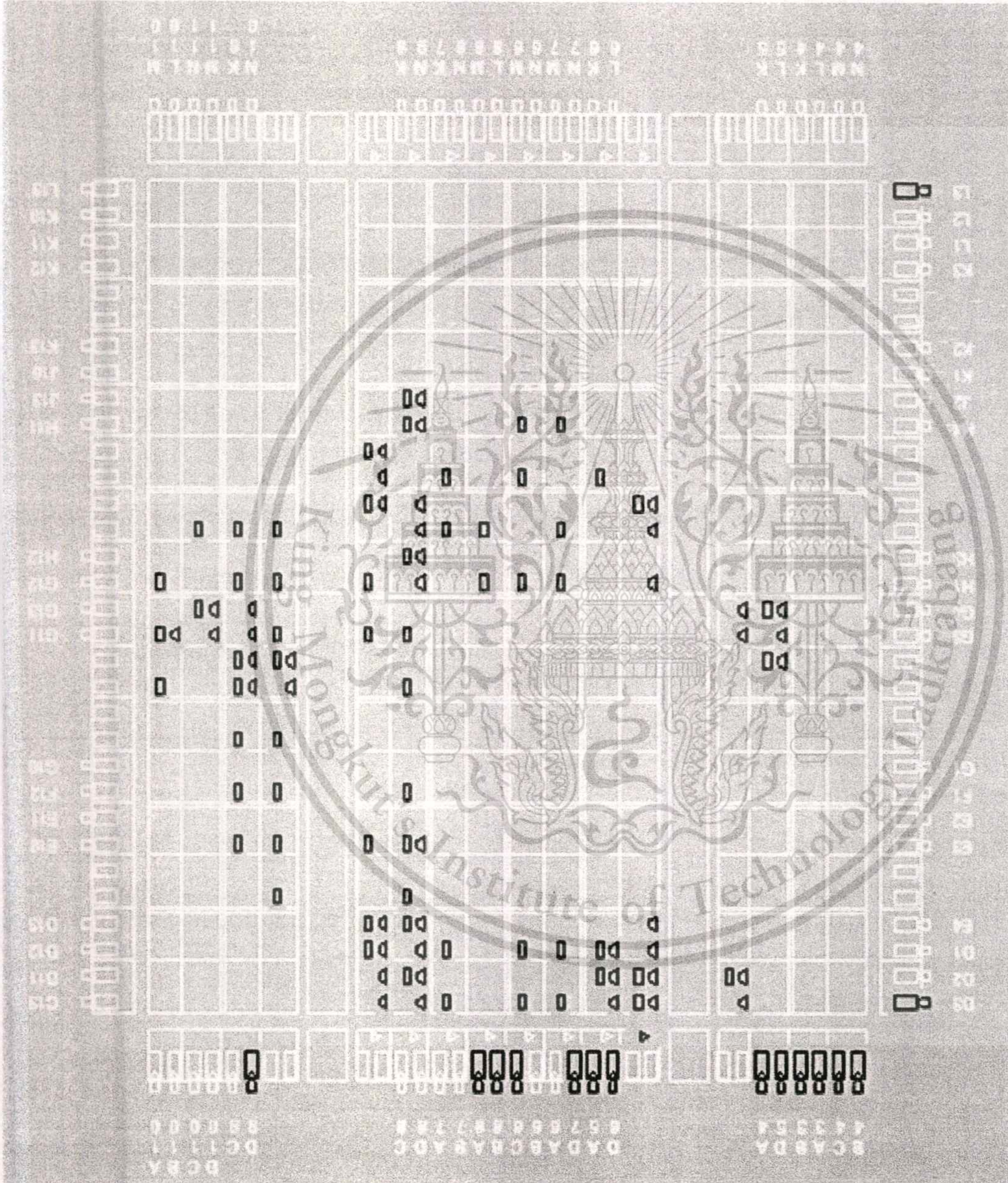


Figure C.4 Floorplan when n = 4
FPGA Family: Xilinx Virtex 2 FPGA
Device: xc2v40

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

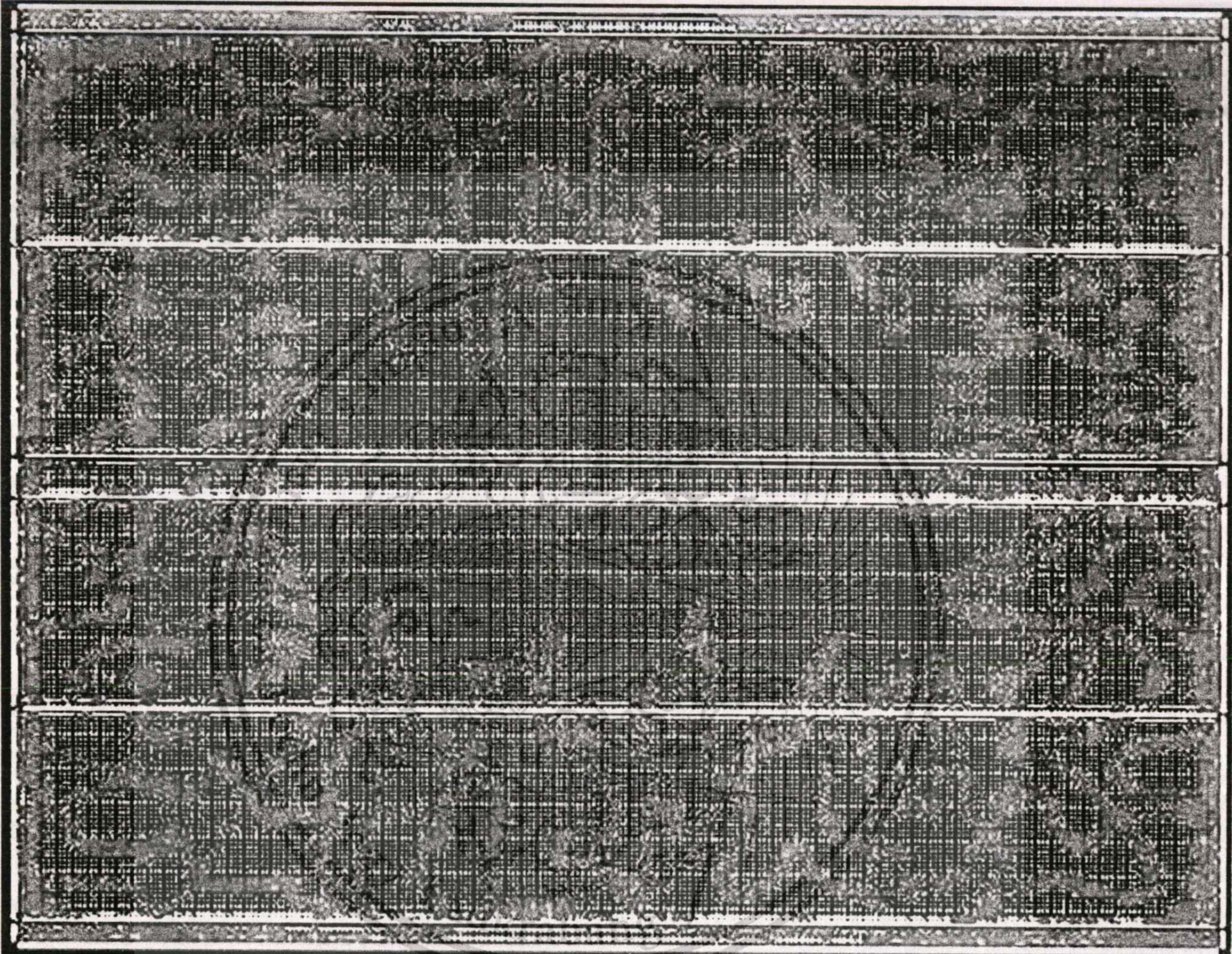


Figure C.5 Floorplan when n = 1024
FPGA Family: Xilinx Virtex 2 FPGA
Device: xc2v8000

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

Appendix D

An FPGA Implementation of Systolic Array for Montgomery Modular Multiplication

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

An FPGA Implementation of Systolic Array for Montgomery Modular Multiplication

Medi A. Nazar, Surin Kittitornkun, and Pratheep Bunyatnoparat

Department of Computer Engineering, Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Email: nazarm@dlsu.edu.ph, kksurin@kmitl.ac.th, kbprathe@kmitl.ac.th

Abstract

Public-key cryptographic algorithms such as RSA algorithm require modular multiplications of very large operands. In RSA, the higher security the larger operand size which may reduce the clock rate and result to lower throughput. This paper presents a fully systolic linear-array for the computation of Montgomery modular multiplication that is implemented using FPGA. Our fully systolic design shows that a high and nearly constant clock rate is achievable regardless of the size of the operand. As compared with the non-fully systolic architecture, our design offers higher frequency that yields a higher throughput rate and a lower area-time product. The total execution time for an n-bit modular multiplication is $3n+1$ cycles, latency of $2n+1$, where n is the length of the modulus and a throughput of 1 bit per clock cycle.

1. Introduction

Secured cryptographic implementation is becoming one of the strong requirements for computer and communication networks, e.g. electronic banking transactions through commercial Internet. Today, hardware-based cryptographic implementation is one of the interests in current research in order to strengthen security and improve performance. For hardware implementations, reconfigurable technology such as the Field Programmable Gate Array (FPGA) offers many advantages over semi-custom Application Specific Integrated Circuits (ASICs). FPGAs assure a short time to market, high flexibility including capability for frequent modifications of hardware, low development cost and low cost of the final product. It has the potential for fast, low cost reprogramming and experimental testing of a large number of various architectures and revised versions of the same architecture. One of the goals of this research is to develop a hardware design for cryptographic algorithm that will be implemented on FPGA.

There are two basic types of cryptographic algorithms; *secret-key* (symmetric) and *public key* (asymmetric). Among the various public-key cryptographic algorithms, RSA (Rivest, Shamir, Adleman) cryptosystem is widely used and considered to be the most popular today [1]. On the other hand, IDEA (International Data Encryption Algorithm) is an example of secret-key algorithm that is very efficient in terms of computational complexity. Public key cryptosystem is based on number theoretic properties, much slower than secret-key but offers higher degree of security.

In RSA, the essential arithmetic operation is modular multiplication, which is used to calculate modular exponentiation on numbers with a size of hundreds of bits. The efficiency requirement in its implementation becomes the motivation in the development of several modular multiplication algorithms and architectures [2-10].

Montgomery algorithm is considered to be the most efficient method in performing series of modular multiplication. C. D. Walter [7] proposed systolic modular multiplication utilizing Montgomery's method that led to various implementation and different techniques on how to further improve its efficiency [7, 8, 9, 10]. Here we adopt the modified Montgomery algorithm done in [8] that is suitable for systolic array.

This research aims to design a fully systolic array for Montgomery multiplication algorithm that is expected to achieve higher frequency which is suitable for higher operand size. We compare and show our improved result with the implementation done in [8].

2. RSA Algorithm

The RSA Algorithm was developed by Ron Rivest, Adi Shamir, and Len Adleman at MIT

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

and first published in 1978. RSA scheme is asymmetric block cipher in which the plaintext and ciphertext are integers between 0 and N-1 for some integer N with a typical size of 1024 bits. Its security is based on the difficulty of factoring large integers. The RSA algorithm requires computation of the modular exponentiation which is broken into a series of modular multiplications. The encryption operation is performed by computing;

$$C = M^e \pmod N$$

where M is the plaintext such that $0 \leq M < N$. The number C is the ciphertext from which the plaintext M can be computed using;

$$M = C^d \pmod N.$$

Hence, e and d had been called as public and private exponents. Here is the binary method for exponentiation where the bits of e are scanned from left to right.

```
Input: M; e; N
Output: C := M^e mod N
1.  if e_{h-1} = 1
      then C := M
    else C := 1
2.  for i = h - 2 downto 0
2a.       C := C * C (mod N)    \\square
2b.       if e_i = 1
            then C := C * M (mod N)  \\ multiply
3.  return C
```

It shows that the modular multiplication and squaring are the main operation used in the above algorithm. The modular multiplication is defined as the computation of $P = AB \pmod N$ given the integers A, B, and N. It is usually assumed that A and B are positive integers with $0 \leq A,B < N$, i.e., they are the least positive residues. There are basically four approaches for computing the product P [11], one of these is Montgomery's method which is suitable for RSA.

2.1 Montgomery Multiplication Algorithm

The Montgomery algorithm computes

$$\text{Montgomery}(A,B) = A * B * r^{-1} \bmod N$$

given $A,B < N$ and r such that $\gcd(N, r) = 1$ (N and radix r are coprime, the greatest common divisor between these two numbers is 1). Even though the algorithm works for any r which is relatively prime to N, it is more useful when r is taken to be a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors. In this paper $r = 2$ and assume that the number of bits in A or B is less than N. Let $A = (A_{n-1} A_{n-2} \cdots A_0)$ be the binary representation of A, $B = (B_{n-1} B_{n-2} \cdots B_0)$ for B and $N = (N_{n-1} N_{n-2} \cdots N_0)$ for modulus N.

Algorithm 2.1 shows how to perform Montgomery modular multiplication by a method of reversing the order of the multiplicand, shifting down instead of shifting up, and adding rather than subtracting multiples of the modulus. Here we are going to adopt the Systolic Montgomery algorithm (*Algorithm 2.3*) of [8].

Algorithm 2.1: Radix-2 Montgomery multiplication algorithm

```
Montgomery(A, B, N)
{         P = 0;
1.        for i = 0 to n-1
2.        {   if (P + a_i B) is even
3.                 P = (P + a_i B) / 2;
4.            else
5.                 P = (P + a_i B + N) / 2 ;
          }
6.        if P ≥ N
7.             P = P - N;
return P ;
}
```

In *Algorithm 2.2*, it can be seen that line 3: $P = (P + a_i B + q_i N) / 2$, is dependent on the values of a_i and q_i, Table 2.1 summarizes its computation where precomputed NB is the sum of N and B. The final subtraction in line 7 is intentionally omitted, because the technique that will be discussed in the succeeding section will concentrate mainly on the iteration involved inside the *for*-loop of Systolic Montgomery algorithm.

Algorithm 2.2: Modified Montgomery multiplication algorithm

```
ModifiedMontgomery(A, B, N)
{         P = 0;
1.        for i = 0 to n-1
2.        {        q_i = (p_0 + a_i b_0) mod 2;
3.                 P = (P + a_i B + q_i N) / 2;
          }
return P;
}
```

Algorithm 2.3 is the modified version of *Algorithm 2.2*, which is more suitable to systolic implementation [8]. The corresponding dependence graph of *Algorithm 2.3* and with the data flowing in it is shown in *Figure 2.1*. All PEs perform the same task except that the right-border PEs have to compute bit q_i as well, see line 2.

Table 2.1: Computation of $P + a_iB + q_iN$ ($NB = N + B$, precomputed)

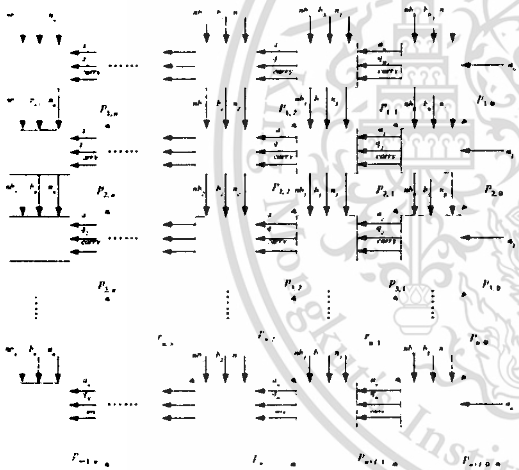
a_i	q_i	$P + a_iB + q_iN$
1	1	$P + NB$
1	0	$P + B$

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

0	1	P + N
0	0	P

Algorithm 2.3: Systolic Montgomery modular algorithm

```
SystolicMontgomery (A, B, N, NB)
{ int P =0;
 bit carry = 0, x;
1.  for i = 0 to n
2.  {  q_i = p_(0,0) ⊕ a_i *b_0;
3.        for j = 0 to n
4.        {  switch a_i, q_i
5.           {  1,1:  x = nb_j
6.              1,0:  x =b_j;
7.              0,1:  x = n_j;
8.              0,0:  x = 0;
           }
9.        p_(i+1),0 = p_(i),(j+1) ⊕ x_i ⊕ carry;
10.       carry = p_(i),(j+1) * x_i  +  p_(i),(j+1) *
                    carry  + x_i * carry;
        }
   }
return P;
}
```



2.1: Dataflow within the Dependence Graph based on Algorithm 2.3.

direction vector $\vec{d}^T = [0\ 1]$ and default schedule vector $\vec{s}^T = [0\ 1]$, while SFG-II in *Figure 3.1c* is the generated SFG when the projection vector $\vec{d}^T$ is [1 0] and schedule vector $\vec{s}^T = [1\ 1]$. The default schedule in the latter SFG is not permissible, thus we chose different vector value to satisfy the two conditions that define the validity of schedule for the given DG. These conditions are:

(1) $\vec{s}^T \vec{d} > 0$.
(2) $\vec{s}^T \vec{e} \geq 0$, for any dependence arc $\vec{e}$.

For the succeeding discussion PE_{IX_i} represents the processing element of DG, $PE_{SFG\text{-}I}$ represents the processing element of SFG-I and $PE_{SFG\text{-}II}$ represents the processing element of SFG-II.

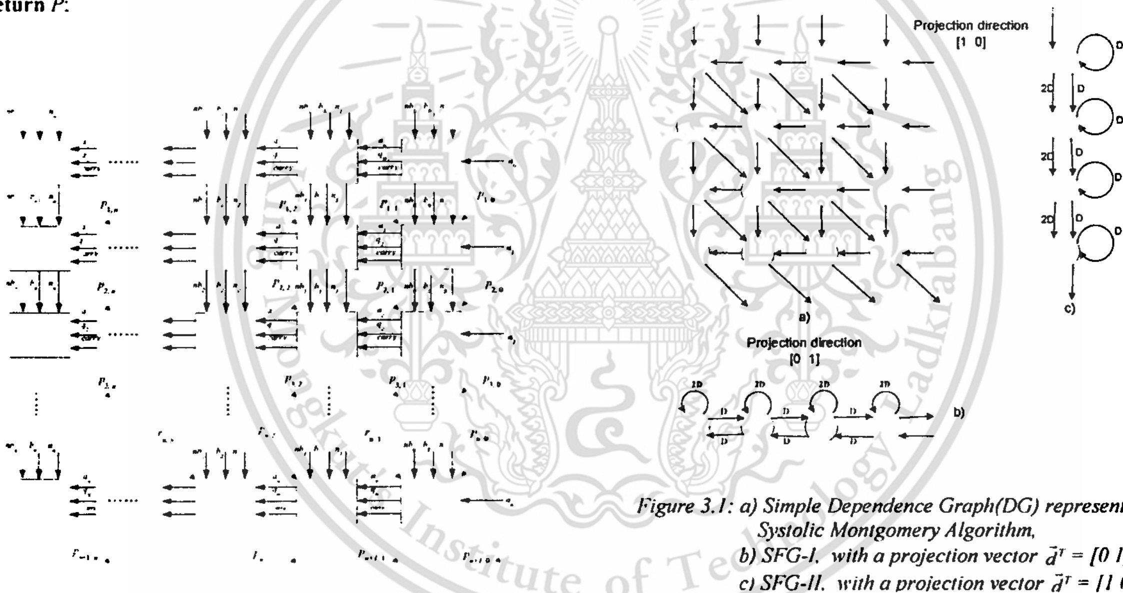


Figure 3.1: a) Simple Dependence Graph(DG) representation of Systolic Montgomery Algorithm,
b) SFG-I, with a projection vector $\vec{d}^T = [0\ 1]$,
c) SFG-II, with a projection vector $\vec{d}^T = [1\ 0]$

All PE_{IX_i}s in each column of DG is projected to each $PE_{SFG\text{-}I}$. Since each PE_{IX_i} of every column of DG performs the same operation, thus the architecture of each $PE_{SFG\text{-}I}$ follows the architecture of the PE_{IX_i}s projected to it. The same thing happens for SFG-II, the PE_{IX_i}s in each row of DG are projected to each $PE_{SFG\text{-}II}$. But this time, since the rightmost PE_{IX_i} in each row of DG performs extra operation for the computation of q_i (see *Algorithm 2.3* and *Figure 2.1*), $PE_{SFG\text{-}II}$ should be designed to accommodate this operation.

3. Systolic Design Methodology

3.1 Mapping DG into SFG

Figure 3.1a is the simple DG representation of *Figure 2.1*. The arrows, the dependence arcs $\vec{e}$, in the same direction are combined into one for the simplicity of the discussion (i.e. the vertical arrow represents inputs nb_i, b_i, and n_i). Each circle represents the processing element, *PE*.

By following the algebraic mapping procedure in [12, 13], two SFGs are derived from DG. SFG-I in Figure 3.1b is the generated SFG by using the projection

3.2 Systolic Mapping

By following the algebraic computation procedure in [12], we're able to systolize the

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

two SFGs in the previous section, *Figure 3.2* shows the systolic SFG-I and SFG-II. It will be shown later that if we incorporate pipelining into an SFG array, it will lead to a systolic design that will give us higher clock rate.

In this paper we used a schedule vector $\vec{s}^T = [1\ 2]$ where the two conditions for selecting schedule vector are satisfied;

$$\vec{s}^T \vec{d} > 0 \text{ and } \vec{s}^T \vec{e} > 0,$$

It guarantees that every edge of the SFG will have one or more delays ($D(e) \geq 1$).

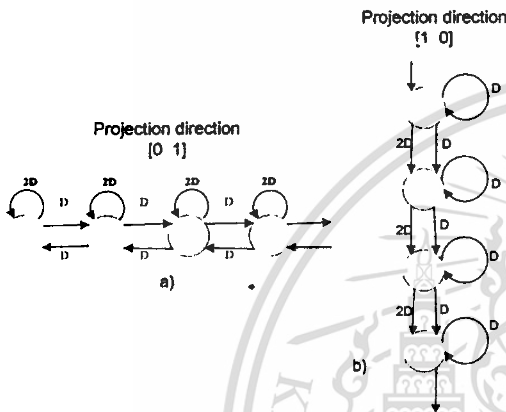


Figure 3.2: a) Systolic SFG-I, b) Systolic SFG-II

For our implementation, we are going to use the systolic SFG-II for two major reasons;

1. *Pipelining period* [12]: the pipelining period of SFG-II is one ($\alpha = 1$), which means that PE utilization is 100%, while pipelining period of SFG-I is two ($\alpha = 2$), PE utilization is reduced to 50%.
2. *Ports*: Based from the algebraic computation, the output ports and the vertical input of SFG-I are located in each node of the array. And in the actual implementation, there are three inputs in a vertical direction, nb_i, b_i, n_i, see *Figure 2.1*, $3(n+1)$ ports are needed for these three inputs where n is the length of the modulus, plus $(n+1)$ for the output ports and a single port for operand A, with a total of $4(n+1) + 1$ ports. In SFG-II there is only one output port, located in the last node of the array. Inputs nb_i, b_i, and n_i will be inputted sequentially in the first node and a single input port at each node for each bit of operand A, this requires a total of $(n+1) + 4$ ports.

4. Simulation and Implementation Results

4.1 Architectural Design in VHDL

We used VHDL in designing the systolic SFG-II, and it is configurable to any size of modulus. The architecture of each $PE_{SFG\text{-}II}$ is shown in *Figure 4.2*. The flip flops FF*i* represent the delays, where *i* is an integer as shown in the figure.

For the top level design, we slightly modified the first and last nodes. In the first node FF0-FF8 is eliminated and there is no F(demux) in the last node. The first node receives the inputs, nb_i, b_i, n_i and *Pi* as well as the *select1* and *select2* signals sequentially at every clock cycle, then propagate through the array. The partial product *P* which is initialized to zero (see *Algorithm 2.3*) will be fed in ports *Pi* and *P0*, LSB at *P0* and the rest of its bits p_1, p_2, ... p_n will flow through *Pi*. Each bit of operand *A* is inputted at every node, a_0 in the first node, a_1 in the second node, and so on, where a_n is always zero. *Pout1* and *Pout0* of a certain node are the *Pi* and *P0* input of the node next to it.

Each bit of modulus *N*, operand *B*, precomputed *NB* and *P0*s that are flowing through the inside nodes have two delays. *P0*s at the inside nodes have two delays each while *Pi*s have one delay. See the delay representation in *Figure 3.2b*.

The last node has only one output, *Pout0*, where the final partial product of the Montgomery multiplication process will be outputted sequentially every clock cycle starting with the least significant bit (LSB).

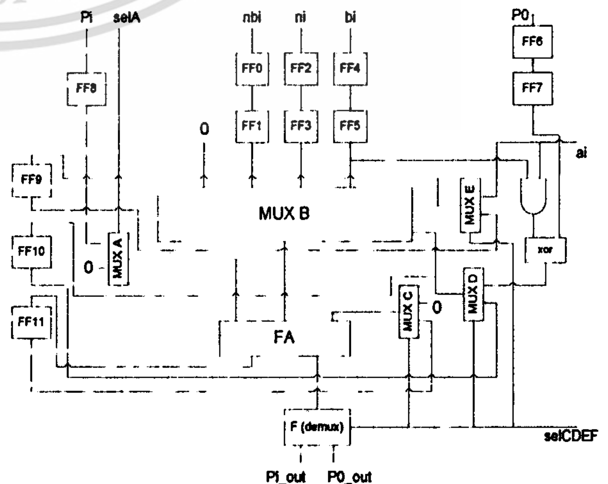


Figure 4.2: Typical structure of each processing element in systolic SFG-II.

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

4.2 Simulation Result

For the simulation and verification, we used Modelsim XE II v5.6a. The funtionality of our design has been verified at n=4, *Figure 4.3* shows the simulation results. The waveform in *Figure 4.3a* shows the output with the following operand values; A = 01100, B = 01100, N = 1111. The first output bit comes out during the ninth clock cycle and the last output bit at 13th clock cycle. *Figure 4.3b* shows the result when A = 01011, B = 01011, N = 1111. It can be seen that the output is greater than N, hence needs a final subtraction, see *Algorithm 2.1*.

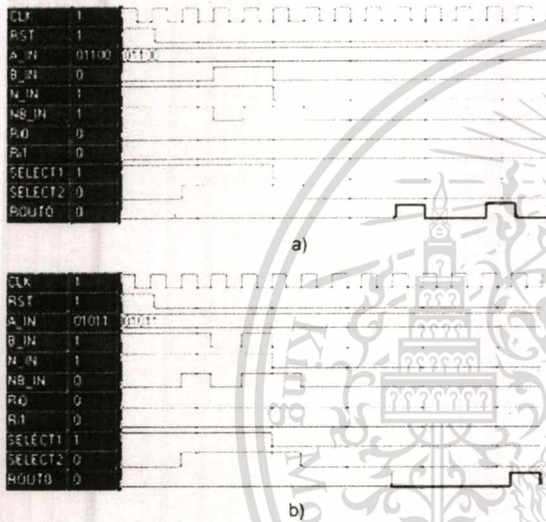


Figure 4.3: Simulation results: a) Output waveform of first simulation. b) Output waveform of second simulation.

From the simulation result, it can be seen that the total execution time is 13 clock cycle, which is equivalent to $3n + 1$ where n, the size of modulus, is 4. The latency is $2n+1$ clock cycles, where the LSB of the final partial product is the first output bit. Then the succeeding bits are outputted sequentially every clock cycle, giving us the throughput of one bit per clock cycle.

4.3 Synthesis and Implementation Result

Our design has been synthesized and implemented using the Xilinx Foundation ISE 5.1i. The implementation device we used is FPGA Virtex2 family. We used different length of modulus during the synthesis and implementation. *Table 4.1* shows the clock cycle time, the area in terms of CLB count and the time-area product for the different modulo size. We also included results from the implementation of [8] for comparison.

Table 4.1: Performance figures, our design vs. [8]

Size of Modulus	Area (CLBs)		Clock cycle time (ns)		Area x Time	
	Ours	[8]	Ours	[8]	Ours	[8]
128	1,026	259	5.424	23	5,565	5,957
256	2,180	304	5.835	42	12,720	12,767
512	4,362	492	5.835	76	25,452	37,392
768	6,545	578	5.835	82	38,190	47,396
1,024	8,729	639	5.835	134	50,934	85,626

As expected, our design utilized a huge amount of CLBs, it is due to the additional multiplexers and flip flops needed for the systolic design. But the clock cycle time of our design is very low, hence higher clock rate (frequency). It is noticeable that the clock cycle time of [8] increase as the size of modulus increases, which is not practical in actual application. While in our design, the clock cycle time is nearly constant, see *Figure 4.4*, resulting to a much lower area-time product as the size of modulo increases.

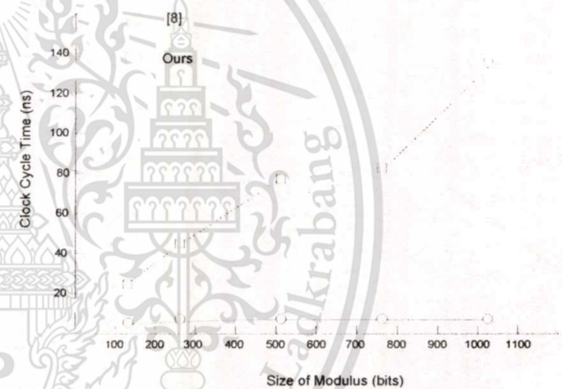


Figure 4.4: Clock cycle time: Nedjah's multiplier vs. Systolic SFG-II multiplier

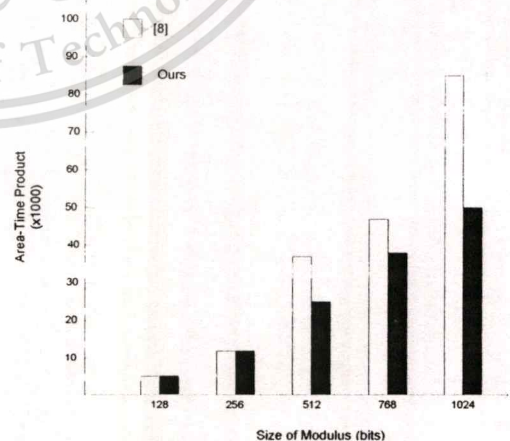


Figure 4.5: Area-Time product: Nedjah's multiplier vs. Systolic SFG-II multiplier

5. Conclusion and Future Works

A fully systolic linear-array for Montgomery modular multiplication is described in this paper. The results obtained showed that

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.

systolic SFG-II achieved a much greater and nearly constant frequency at different size of modulus which is very attractive in actual application, because in RSA, the higher the number of bits the more security it has, which needs longer time during exponentiation. This high frequency in turn leads to a higher throughput. The systolic SFG-II is implemented in FPGA which inherent all its flexibility features. This is desirable for this kind of design since security requirement increases with time thus requiring for future developments.

Minimizing IOBs is important for this design because FPGAs has limited numbers of IOBs. In our systolization process, we chose the SFG derived using the projection direction $d' = [1\ 0]$, yielding a lower number of IOBs and 100% PE utilization compare to the SFG derived using the projection direction $\vec{d}^T = [0\ 1]$ that requires more IOBs which may further require for more than one FPGAs for its implementation. We utilized one FPGA Virtex2 family with enough IOBs for 1024-bit modulus.

This multiplier array can be used in the exponentiation operation which is a series of modulo multiplication. Our multiplier array can be reused for the next multiplication right after the first bit (LSB) of the current multiplication is outputted, by feeding it back to the array. Thus, 100% utilization can be achieved during the entire exponentiation as discussed in Section 3.2.

6. Acknowledgments

The authors would like to thank ASEAN University Network/Southeast Asia Engineering Education Development Network (AUN/SEED-Net) for the financial support of this research and also to the active collaboration of the coordinators between the Host Institution (KMITL) and Sending Institution (DLSU) of the said organization.

7. References

[1] B. Schneier, *Applied Cryptography Second Edition*, John Wiley & Sons, N. Y., 1996.

[2] Y. J. Jeong and W. P. Burleson, *VLSI Array Algorithms and Architectures for RSA Modular Multiplication*, IEEE Transactions on VLSI Systems, Vol. 5, No. 2, pp. 211-217, June 1997.

[3] J. Groszschaedl, *The Chinese Remainder Theorem and its Application in a High-Speed RSA Crypto Chip*, Computer Security Applications, 2000, ACSAC '00, 16th Annual Conference, Dec. 2000.

[4] T. Blum and C. Paar, *High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware*. IEEE Transaction on Computers, Vol. 50, No. 7, pp.759-764, July 2001.

[5] A. Daly and W. Marnane, *Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic*, FPGA'02, Monterey, CA, pp. 40-49, Feb. 2002.

[6] A. F. Tenca and C. K. Koc, *A Scalable Architecture for Modular Multiplication based on Montgomery's Algorithm*, IEEE Transactions on Computers, Vol. 52, No. 9, pp.1215-1221, Sept. 2003.

[7] C. D. Walter, *Systolic modular multiplication*, EEE Transactions on Computers, Vol. 42, No. 3, pp. 376-378, March 1993.

[8] N. Nedjah and L. M. Mourelle, *Two hardware implementations for the Montgomery multiplication: sequential vs. parallel*, Proc. of the 15th. Symposium on Integrated Circuits and Systems Design, Porto Alegre, RS, Brazil, IEEE Computer Society Press, pp. 3-8, 2002.

[9] P. Kornerup, *A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms*, IEEE Transaction on Computers, Vol. 43, No. 8, pp. 892-898, Aug. 1994.

[10] J. H. Hong and C. W. Wu, *Cellular-Array Modular Multiplier for Fast RSA Public-Key Cryptosystem Based on Modified Booth's Algorithm*, Vol. 11, No. 3, pp. 474-484, June 2003.

[11] C. K. Koc, *RSA Hardware Implementation*, RSA Labs Technical Report TR-801 v1.0, RSA Data Security, Inc., CA, August 1995.

[12] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, Englewood Cliffs, N.J., 1988.

[13] H. T. Kung, *"Why Systolic Architectures?"*, IEEE Computer Magazine, Vol. 15, No. 1, pp. 37-46, Jan. 1982.

This material is reserved for educational use only, not allowed for commercial use.

Forbidden to modify the content, and cite the document when use.

Author's Biography

Miss Medi A. Nazar was born on January 7, 1978 in San Miguel, Bulacan, Philippines. She received her bachelor's degree in Electronics and Communications Engineering (ECE) with honor from the University of Santo Tomas, Manila in 2000. She worked in ROHM LSI Design Philippines, Inc. before she took her licensure examination on the same year. After passing the examination, she worked at De La Salle University (DLSU), Manila as academic service faculty at ECE Department and took courses in graduate studies while working there. In 2002, she has been granted an scholarship from ASEAN University Network/Southeast Asia Engineering Education Development Network (AUN/SEED-Net) for a master's program here in King Mongkut's Institute of Technology Ladkrabang (KMITL). She will be working again with ECE Department of DLSU on January 2003. Her research interests include computer arithmetic, digital hardware design, and cryptography.

King Mongkut's Institute of Technology Ladkrabang

This material is reserved for educational use only, not allowed for commercial use.
Forbidden to modify the content, and cite the document when use.