TK
7874.65
P68
2595

A HARDWARE BASED TECHNIQUE TO REDUCE THE TIMING COMPLEXITY OF NUMBER FACTORING PROBLEM

By

Pirouz Pourdowlat

Bachelor of Computer Engineering

University of Toronto

Toronto, Canada, 2003

A Thesis

Presented to Ryerson University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada

© Pirouz Pourdowlat 2006

UMI Number: EC53530

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform EC53530
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature

ABSTRACT

Title of Thesis: **A Hardware Based Technique to Reduce the
Timing Complexity of Number Factoring
Problem**

Pirouz Pourdowlat, Optimization Problems Research and Applications Laboratory
(OPR-AL), ELCE, Master of Applied Science, Ryerson University 2006

Thesis Directed by: Dr. Nagi Mekhiel and Dr. Reza Sedaghat,
Department of Electrical and Computer Engineering

Most digital circuits which have been developed to implement algorithms, can benefit from an increase in clock speed, but do not completely map the problem to all available silicon resources. We have introduced a hardware based scheme capable of effectively using technology, specifically the increase in silicon area, to improve the computational time of complicated applications. In this thesis, we applied this scheme to solve the factoring problem, which requires exponential time (with respect to the number of bits in n) in conventional computers and could be only solved in polynomial time with quantum computers. The scheme successfully mapped the problem to most of the silicon area of Altera Stratix FPGA. The results show that the scheme is capable of reducing the time complexity to a polynomial rate with respect to the number of bits of the number n . The results also show an exponential rate of use for silicon with respect to the number of bits of n . Our analysis shows that the new scheme is scalable with technology speed and

available space, could be applied to other applications to solve the performance limitations of conventional systems.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mekhiel and Dr. Sedaghat for being my supervisor throughout my graduate studies. Their comments and suggestions were invaluable to the improvement of this thesis.

I am grateful to OPR-AL lab and members for their contribution and support.

TABLE OF CONTENTS

ABSTRACT.....	iv
ACKNOWLEDGEMENTS	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	ix
LIST OF FIGURES	x
NOMENCLATURE.....	xiii
CHAPTER 1 Introduction	1
1.1 Overview	1
1.2 Contributions.....	3
1.3 Organization of Thesis.....	4
CHAPTER 2 Background.....	6
2.2 Shor's Algorithm and Quantum Computers.....	7
2.2.1 Some Basic Concepts in Quantum Computers	8
2.2.1.1 From Classical Bits to Quantum Bits (Qubits).....	8
2.2.1.2 State Vectors and Superposition.....	10
2.2.1.3 Unitary Operators.....	12
2.2.2 Shor's Algorithm.....	15
2.2.2.1 How Shor's Algorithm Works	16
CHAPTER 3 The Concept and Motivations for the Hardware Based Technique..	22
3.1 The Main Idea	22
3.2 Logic Function of m Inputs.....	23
3.3 Algebraic Functions	25
3.4 Calculation of the Area Under a Curve $f(x)$	27
3.5 Vector Multiplication.....	29
CHAPTER 4 Application of 'IT' to the Factoring Problem.....	31
4.1 Architectural Overview	31
4.2 First Stage	35
4.3 Second Stage	40
4.4 A Remark Regarding the Time Delay Formula.....	45
4.5 Third Stage	46
CHAPTER 5 A New Circuit to Reduce the Complexity of Searching.....	48
5.1 Complexity of Search Algorithms in General	48
5.2 The Proposed Search Algorithm, Implemented in Hardware.....	49
5.3 An Important Note Regarding the Timing of this Search Scheme	55
5.4 A Final Remark.....	57
CHAPTER 6 The Rate of Growth in the Minimum Clock Period.....	58
6.1 Importance of the Issue	58
6.2 Delay of a Combinational Logic Unit in General.....	59
6.3 Timing Delay of a Multiplier of the Size m by m	60
6.4 Timing Complexity of a Divider of the Size $2m$ -bit by m -bit.....	62
6.5 Maximum Clock Frequency of a FSM.....	63
6.6 The Over-all Timing Complexity of the Circuit.....	75

CHAPTER 7 Area Complexity of the Circuit.....	77
7.1 The Order of the Rate of the Growth of Area.....	77
CHAPTER 8 Experimental Results	81
8.1 Methodology	81
8.2 Results	83
CHAPTER 9 Conclusions and Future Works.....	93
9.1 Conclusions.....	93
9.2 Future Works	94
REFERENCES:	96

LIST OF TABLES

Table 8.1 Device features for Stratix EP1S40F780C5

LIST OF FIGURES

Figure 2.1: DFT of $x_1[n] = [3 \ 7 \ 6 \ \dots 3 \ 7 \ 6]$ (length of the sequence=81)

Figure 2.2: DFT of $x_2[n] = [3 \ 7 \ 6 \ \dots 3 \ 7 \ 6]$ (length of the sequence=729)

Figure 2.3: DFT of $x_3[n] = [1 \ 12 \ 2 \ 7 \ 4 \ \dots 1 \ 12 \ 2 \ 7 \ 4]$ (length of the sequence=25)

Figure 2.4: DFT of $x_4[n] = [1 \ 12 \ 2 \ 7 \ 4 \ 1 \ \dots 1 \ 12 \ 2 \ 7 \ 4]$ (length of the sequence=3125)

Figure 2.5: DFT of $x_5[n] = [1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \dots 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17]$ (length of the sequence=100)

Figure 2.6: DFT of $x_6[n] = [1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \dots 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17]$ (length of the sequence=10000)

Figure 3.1: The IT applied to finding the value of a logic function with 4 inputs

Figure 3.2: The application of IT to the Problem of Calculating an 8th Degree Polynomial

Figure 3.3: Using IT to Calculate $\int_a^b f(x)dx$

Figure 3.4: The IT applied to the problem of calculation of $C_{11} = A_{1xn}XB_{nx1}$

Figure 4.1: The Block Diagram for the over all System (4-Bit Wide), in This Example $n=15$ and $x=4$

Figure 4.2: Datapath for the First Stage (Upper Part)

Figure 4.3: Timing diagram for the output signals of the upper part controller

Figure 4.4: The lower part of a 5-bit system

Figure 5.1: An example of the search for the number '8' among a set of eight numbers: $\{9,5,3,8,8,7,8,2\}$

Figure 5.2: An example of the simultaneous comparison of the key number to a set of numbers followed by the binary tree of 'OR' gates

Figure 5.3: The state transition diagram of the FSM to traverse the tree in figure 5.2

Figure 5.4: The 'OR' gate tree to be traversed for a general case of searching among 'n' numbers

Figure 6.1: The tree of multiplexers to implement any function F

Figure 6.2: A model for Finite State Machine (FSM)

Figure 6.3: Main components of the divider data path

Figure 6.4: The block diagram for the FSM in the search circuit

Figure 6.5: Details of the tree of 2 to 1 multiplexers (the block at the bottom of figure 6.4)

Figure 6.6: The tree of multiplexers for a search FSM with 4 state bits

Figure 6.7: The tree of OR gates to be traversed for a case with 4 state bits

Figure 8.1: The search algorithm finds the position of the least significant '1', in the pattern: $2^{38} + 2^{41} + 2^{68}$

Figure 8.2: As the number of bits in 'n' increases by one, the amount taken for the calculation of 'r' increases by a roughly constant amount

Figure 8.3: The logarithmic rate of increase in the time required to calculate the order of $x(\text{mod } n)$, is further verified, by using pairs of x and n different from the one used in figure 8.2

Figure 8.4: Time delay for the calculation of 'r' increases linearly with the number of bits in 'n'

Figure 8.5: Timing details of calculation the period of $f_n(a) = x^a(\text{mod } n)$, for $n=15$ and $x=13$

Figure 8.6: Timing details of calculation the period of $f_n(a) = x^a \pmod n$, for $n=209$ and $x=13$

Figure 8.7: The growth of area as the system width increases from 4 to 8 bit

Figure 8.8: Minimum clock period (in ns) as the system width increases from 4 to 8 bit

NOMENCLATURE

R	Set of Real numbers
C	Set of Complex numbers
N	Set of Natural Numbers
Z	Set of Integers
DFT	Discrete Fourier Transform

CHAPTER 1

INTRODUCTION

1.1 Overview

Technology has made it possible to produce integrated circuits whose number of transistors is in the order of hundreds of millions. In fact since the early 1960's the rates at which the die size and the number of transistors per unit area increases have been exponential [1,2,3]. Furthermore, despite the fact that this exponential rate of increase (both in the density and the die size) can not be sustained forever [1], innovative ideas such as double-gate [4] and triple-gate [5] structures for transistors, fully-depleted silicon-on-insulator [6] and modification of the crystal structure of the silicon to increase carrier mobility [7] make us believe that this exponential rate can be sustained for several more generations. As a result of this fast rate of progress, it is possible to say that the challenge is no longer to produce chips containing millions of transistors but it is how to best use devices containing such chips [8]. Although the demand for more computing power is growing the huge advancements in creating denser and bigger circuits on silicon have not been fully utilized to produce more computing power. The performance of uniprocessor systems is limited by the memory bandwidth and latency [9], communication between the different parts of the program could limit the performance of the parallel processors (in the particular case of factoring problem, the fixed number of processors in a parallel processing system is the main bottleneck). Many research works have proposed new architectures to deal with the limitations of the current processing

models. The IRAM project integrates the processor and the DRAM in the same die to expose the processor to the high DRAM bandwidth [10]. IRAM performance depends on finding parallelism at the instruction level through vector operations. The RAW project uses compiler technology to fully expose hardware, including wiring, to system software. The software can use the abundance of logic gates and reroute information through the reconfigurable wires to provide a scalable gate and wire delay [11]. The M-Machine Project is a fine grained multi-computer that exploits the circuit density. M-Machine uses multi-ALU processing with minimum communication overhead [12] to improve the performance. All these projects improve the performance by using parallelism at the instruction level, wire delay level, or processor level. In this thesis an idea for a new hardware based technique (Integration Technique) is introduced, and one of its possible applications is shown in detail. The particular application in question finds the order of 'x modulo n' in logarithmic time (with respect to the size of 'n'). Finding the value of the order of 'x modulo n' in logarithmic time could be in turn used to solve the problem of factoring numbers in logarithmic time. All the known algorithms for factoring on conventional computers have exponential run-times (with respect to the number of bits in the number to be factored), even a multiprocessor system can only map the problem to a fixed N number of processors and must then deal with the problem of communication between the various parts of the program, Integration Technique (IT) however, maps the problem to all the silicon resources available.

1.2 Contributions

The purpose of this thesis is to introduce the idea of a new hardware scheme (IT) which exploits the increase in die size and transistor density per unit area directly in reducing the timing complexity of complicated factoring algorithm. In this thesis we give the details of application of IT to the factoring problem, together with all necessary theoretical proofs and experimental results (confirming the theory). Although the same idea (IT) could be applied to various algorithms/calculations, since we have implemented it only for the factoring problem, in this thesis we only explain some other possible *candidates* for application of this idea. The over all research contributions of this thesis could be summarized as follows:

- The introduction of the idea of a new hardware technique (Integration Technique) which is based on exploiting the vast amounts of the available silicon to reduce the timing complexity of algorithms at the cost of increasing the silicon area consumption complexity.
- A hardware implemented search scheme, capable of finding the first occurrence of a number from an ‘unsorted’ set of numbers in logarithmic time (with respect to the number of elements in the set).
- The application of the Integration Technique to the problem of finding the order of ‘ x modulo n ’, using the hardware implemented search scheme mentioned earlier.

- Analyzing time complexity versus silicon area complexity for implementation of the complicated factoring algorithm
- Implementation of IT applied to the problem of finding the order of 'x modulo n', in FPGA to verify the technique
- Investigating possible generalization of scheme for other applications

1.3 Organization of Thesis

The organization of this thesis is as follows: Chapter 2 gives background information about the number factoring problem, the time complexity of the existing algorithms and also some information regarding the quantum computers and Shor's approach to the number factoring problem. Chapter 3 introduces the basic idea of IT and three very simple examples which could be considered as *possible candidates* where this scheme could be applied. Chapter 4, explains the details of application of IT to the problem of number factoring. Chapter 5 details a search scheme which is used in the IT approach to solve the factoring problem. Chapter 6 explains the derivation of the formula for the rate of growth in the minimum clock frequency as the width of the architecture (in bits) grows. Chapter 7 discusses the rate of growth in the silicon area occupied by our system as the width of the architecture (in bits) grows. Chapter 8 gives the details of the experimental results and finally, Chapter 9 summarizes the thesis, discusses the results and the possible future works.

Please note that from this point on unless otherwise is mentioned $\log$ means logarithm in the basis of 2.

CHAPTER 2

BACKGROUND

2.1 Factoring Problem

The security of public key cryptosystems is based on the assumption that factoring large numbers is a difficult problem. Currently the typical RSA keys are 1024–2048 bits long. While it is argued that 1024-bit keys maybe broken in foreseeable future, experts seem to agree that there is no prospect of breaking 4096-bit keys with the *conventional* computing systems. The reason for the difficulty of breaking RSA keys is that although there is no rigorous proof, factoring is considered to be a hard problem [13]. This is mainly based on the fact that we have as yet not found an algorithm to run on a conventional computer which would be capable to factor numbers in polynomial time with respect to their number of digit. There are a variety of algorithms proposed to do factoring on conventional computers. Some of these algorithms and their corresponding expected run-times are [14, 15, 16, 17]:

- The Quadratic Sieve Algorithm, $\exp(\sqrt{\log(n) * \log(\log(n))})$
- The Continued Fraction Algorithm, $\exp(\sqrt{2 * \log(n) * \log(\log(n))})$
- The Elliptic Curve Algorithm, $\exp(\sqrt{\log(n) * \log(\log(n))})$

However, the fastest algorithm developed for conventional computers is Number Field Sieve which has an expected run time of: $\exp((c + o(1)) * \log(n)^{1/3} * \log(\log(n))^{2/3})$ [18], where ‘c’ is a constant whose value is approximately 1.526.

As it could be seen, all of these algorithms (although, none has been rigorously proven and as such these are the “heuristically estimated expected running times”) have runtimes which are exponential with the number of digits in the number to be factored, although in all these algorithms the exponent of the exponential has various powers of less than 1 (i.e. they are all sub-linear with the number n to be factored). The only algorithm so far which has had a complexity polynomial with the number of digits in the number to be factored is Shor’s algorithm which could be only run on a quantum computer and the quantum computers are still more at a theoretical stage [19].

The factoring scheme proposed in this thesis has a logarithmic run time and it is based on the same mathematical principles as Shor’s algorithm. For this reason it will be very useful to explain the main principles of Shor’s algorithm and the quantum computers.

2.2 Shor’s Algorithm and Quantum Computers

The main goal of Shor’s algorithm is to take a large integer which is the result of the multiplication of two very large prime numbers, as its input and find the original prime number factors whose multiplication resulted in the large integer which was input to the algorithm.

Shor’s algorithm can solve this problem in $O((\log(n))^2 * \log(\log(n)))$ on a quantum computer plus $O(\log n)$ steps of post processing on a regular computer [19].

Since Shor’s algorithm has been designed to run on a quantum computer, it is impossible to understand his algorithm unless the reader is familiar with some of the basic principles

of the quantum computers. Therefore, here, we first give a brief explanation of the basic concepts underlying quantum computing and then proceed to give an explanation of each step in Shor's algorithm.

2.2.1 Some Basic Concepts in Quantum Computers

In 1980 Paul Benioff suggested a reversible Turing machine which incorporated certain quantum mechanical interactions in some of its operations. This was the first hint to the birth of a new computing based on concepts of quantum mechanics. In 1982 Richard Feynman showed that no classical computer could simulate an arbitrary quantum system efficiently. He further suggested that only a computer based on a quantum system would have a quantum nature and therefore would be capable of simulating an arbitrary quantum system. These were the first major steps in the birth of quantum computing [20]. In the upcoming subsections we will give a basic overview of the main components of the quantum computers and explain how they differ from their classical counterparts and the implications of these differences.

2.2.1.1 From Classical Bits to Quantum Bits (Qubits)

First let's reflect on the definition of a conventional bit and its storage (a classical register). Bit which is an abbreviation for Binary Digit is the smallest unit of digital information in a computer. It can ONLY have one of the two values of '0' and '1'. In a conventional computer, bits are stored in registers. A register is usually made of a string of flip-flops which are grouped together to store a number of bits (e.g. a byte, a word, etc.). How one stores the one bit is as follows: an entity which can have two possible

states (a flip-flop in case of conventional computers), say state A and state B, is taken to be the storage of one bit. By convention if '1' is to be stored the entity is set into state A and if '0' is to be stored the entity is set to the state B. In case of a conventional computer states A and B are voltage levels. Naturally the storage for a bit in a conventional computer can only be in one of the two states of A or B. Therefore, to store an integer of N, $\log N$ number of flip-flops will be needed or in other words, given m flip-flops only m bits can be stored at any given instant in time.

By contrast quantum computers use any type of quantum system in the nature as their basis for storing a bit of information. In quantum physics, the state of a quantum system is described by a vector in Hilbert space [20]. A Hilbert space is an inner product space that is complete with respect to the norm defined by the inner product. The basis vectors for the Hilbert space of the vectors defining the states of the quantum system are called "eigenstates". Such a system when observed can be found in one of the finite number of states, called "eigenstates" [20]. For example, let's say that the spin state of a spin-1/2 particle is taken to represent the binary information of '1' or '0'. Then *when observed* the particle can be in either of the two spin state, and based on whether it has 1/2 or -1/2 spin, we *measure* a '1' or a '0'. However, there is a major difference between the quantum system and the conventional system (flip-flop) in that while a flip-flop could be either in the state of '1' *or* in the state of '0', a quantum system could be in a "superposition" of its eigenstates. The fact that a quantum system is in a superposition of its eigenstates, means that associated with each eigenstate, there is a calculable possibility that when *observed*, the quantum system maybe found in that particular eigenstate. For example if the quantum system is a spin-1/2 particle, then when the particle is not observed, it is in a

linear combination of the two states of $|1/2\rangle$ and $|-1/2\rangle$, and there is a calculable probability associated with each of these two states. This probability signifies the likelihood of finding the quantum state in the corresponding state *if it is observed*.

2.2.1.2 State Vectors and Superposition

As it was mentioned before, a quantum systems state is described with a space vector in the Hilbert space. The basis for this space is the eigenstates (states which represent the digital information, stored in the quantum register). The state of the system is represented by a linear combination of the basis vectors where the coordinates are complex numbers. These coordinates are of extreme significance, as each coordinate's amplitude is proportional to the square root of the probability of finding the system in that eigenstate if a measurement is performed. Imagine a system with two eigenstates (namely $|\Psi_0\rangle$ and $|\Psi_1\rangle$), and whose state is represented as follows:

$$|\Psi\rangle = w_0 * |\Psi_0\rangle + w_1 * |\Psi_1\rangle \text{ where } w_0, w_1 \in \mathbb{C} \quad (2.1)$$

If a measurement to this system is done there is a probability of:

$$p = \frac{|w_0|^2}{(|w_0|^2 + |w_1|^2)} \quad (2.2)$$

that we may find the system in the eigenstate $|\Psi_0\rangle$ and a probability of:

$$(1-p) = \frac{|w_1|^2}{(|w_0|^2 + |w_1|^2)} \quad (2.3)$$

that we may find it in the eigenstate $|\Psi_1\rangle$ [20].

Note that there are only two possible states to find the system in when we *measure* it:

$|\Psi_0\rangle$ and $|\Psi_1\rangle$.

The situation mentioned above could be generalized to systems whose dimension is bigger than 2 [20]:

$$|\Psi\rangle = \sum (w_i * |\Psi_i\rangle), \quad \text{where } i=0,1,\dots,n-1 \text{ and } n \text{ is the dimension of the Hilbert space} \quad (2.4)$$

The state vector could be easily expressed with matrix notation, for example in case of 2 dimensional Hilbert space, assuming that:

$$|\Psi_0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |\Psi_1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.5)$$

We can express the state vector as:

$$|\Psi\rangle = w_0 * \begin{bmatrix} 1 \\ 0 \end{bmatrix} + w_1 * \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \quad (2.6)$$

The fact that in any particular instant in time an *unobserved* quantum system is in a linear combination of its eigenstates, is referred to as “superposition” [20]. In other words, one can say that at any arbitrary instant in time an unobserved quantum system is in a superposition of its eigenstates. This means that a quantum register of m qubits, assuming that each qubit has only 2 eigenstates, could store a combination of 2^m bits simultaneously. If we take such a system as a quantum register consisting of m qubits then the state vector of such a system could be expressed as the linear combination of a set of new basis vectors obtained by the direct multiplication of the old basis vectors. The case for a register of 2 qubits is as follows:

$$|\Psi_{(1)}\rangle \times |\Psi_{(2)}\rangle \quad (2.7)$$

where the numbers in the parentheses indicate that the two state vectors originally belong to two different Hilbert spaces, and the $\times$ is the symbol for a direct multiplication.

$$|\Psi_{(1)}\rangle \times |\Psi_{(2)}\rangle = \begin{bmatrix} w_{0(1)} \\ w_{1(1)} \end{bmatrix} \times \begin{bmatrix} w_{0(2)} \\ w_{1(2)} \end{bmatrix} = \begin{bmatrix} w_{00} \\ w_{01} \\ w_{10} \\ w_{11} \end{bmatrix} \quad (2.8)$$

Hence the new basis for the composite system is:

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \text{ and } |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (2.9)$$

2.2.1.3 Unitary Operators

Unitary operators are the quantum computer equivalent of the logic gates in the classical computers. In quantum mechanics the mathematical description of how something changes in the quantum system is the application of a unitary operator. Basically unitary operator is a square matrix which is multiplied by the state vector of the system and in so doing it changes the old state vector and creates a new one with its own new set of probability amplitudes. It is fairly obvious that any unitary operator which works on a quantum system consisting of a combination of n number of 2-state systems (such as a quantum register consisting of two qubits, each of which has two eigenstates) is actually a

$2^n \times 2^n$ square matrix. So a unitary operator of a single qubit would be a 2×2 square matrix. An example of a unitary operator is a NOT operator:

$$NOT \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (2.10)$$

Therefore:

$$NOT|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \quad (2.11)$$

Similarly it could be shown that $NOT|1\rangle = |0\rangle$ [20].

In classical logic design a set of logic gates is said to be universal if any combinational logic function can be built from them. A good example would be the set of {NAND, NOT} from which any combinational function could be built. Similarly “universal” operators could be built for quantum computing. One example is the following which works on a 2-qubit system [20]:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{(i\alpha)} \cos(\theta) & -ie^{(i(\alpha-\varphi))} \sin(\theta) \\ 0 & 0 & -ie^{(i(\alpha+\varphi))} \sin(\theta) & e^{(i\alpha)} \cos(\theta) \end{bmatrix} \quad (2.12)$$

where φ , α and θ are constant irrational multiples of π and each other.

It should be somewhat apparent by now that if we have a register (3 qubit long) in the state of:

$$|\Psi\rangle = w^*|\Psi_0\rangle + w^*|\Psi_1\rangle + w^*|\Psi_2\rangle + w^*|\Psi_3\rangle + w^*|\Psi_4\rangle + w^*|\Psi_5\rangle + w^*|\Psi_6\rangle + w^*|\Psi_7\rangle \quad (2.13)$$

meaning that all the eigenstates are equally probable, then applying a function $f()$ to the register, we simultaneously calculate $f(|\Psi_i\rangle)$ for $i=0,\dots,7$ all in parallel, achieving an exponential rate of speed up. And the final state of the register would be:

$$f(|\Psi\rangle) = w * f(|\Psi_0\rangle) + w * f(|\Psi_1\rangle) + w * f(|\Psi_2\rangle) + w * f(|\Psi_3\rangle) + w * f(|\Psi_4\rangle) + w * f(|\Psi_5\rangle) + w * f(|\Psi_6\rangle) + w * f(|\Psi_7\rangle). \quad (2.14)$$

This is the main idea behind the concept of quantum parallelism. However, there is an important catch: If $f(|\Psi_i\rangle)$ and $f(|\Psi_j\rangle)$ are orthonormal for all i and j from 0 to 7, then when we measure the register we will end up obtaining $f(|\Psi_i\rangle)$ ($0 \leq i \leq 7$) with equal likelihood. Therefore the only way for us to make a meaningful observation to calculate the result a particular $f(|\Psi_{i_0}\rangle)$ is to make $f(|\Psi_i\rangle)$'s to interfere with each other (to have components along each other) and in so doing some $f(|\Psi_i\rangle)$ s and $f(|\Psi_j\rangle)$ s may have a destructive interference on each other and we may end up different probability weights for different $f(|\Psi_i\rangle)$ s. So for example in the $f(|\Psi\rangle)$ given above, if we are interested in measuring the value of $f(|\Psi_2\rangle)$, then if all $f(|\Psi_i\rangle)$'s are orthonormal to each other (i.e. the inner product of any $f(|\Psi_i\rangle)$ and $f(|\Psi_j\rangle)$ is equal to 0) we won't be able to measure the value of $f(|\Psi_2\rangle)$, because all $f(|\Psi_i\rangle)$'s are equally likely to be observed (they all have a weight of 'w'). However, if, for example

$$\|f(|\Psi_0\rangle) + f(|\Psi_1\rangle)\| < 1 \text{ and}$$

$$\|f(|\Psi_3\rangle) + f(|\Psi_4\rangle) + f(|\Psi_5\rangle) + f(|\Psi_6\rangle) + f(|\Psi_7\rangle)\| = 0,$$

then we have a very high probability of observing the value of $f(|\Psi_2\rangle)$ when we measure the quantum register. In fact the probability of observing the value of $f(|\Psi_2\rangle)$ is

$$P(\text{observing } f(|\Psi_2\rangle)) = \frac{w}{w + w\|f(|\Psi_0\rangle) + f(|\Psi_1\rangle)\|} \approx 1$$

As a result any algorithm written for quantum computers has to somehow manipulate the “superposition” of eigenstates in the quantum register, in a way that the probability of the observation of the intended “result” will be very high. This subject will become clearer in the next section as we explain the steps involved in Shor’s algorithm.

2.2.2 Shor’s Algorithm

Shor’s algorithm makes use of a result from number theory that shows that for any given integer n , if an x coprime to n is chosen (i.e. $\gcd(x,n)=1$) then the function:

$$f_n(a) = x^a \bmod n \quad (a \in \mathbb{Z})$$

is periodic[21]. Therefore:

$$f(0)=1, f(1)=x, \dots f(r)=1, f(r+1)=x, \dots \text{ for some value of } r.$$

Hence one can write:

$$x^r \bmod n = 1 \Rightarrow (x^{\binom{r}{2}})^2 \bmod n = 1 \Rightarrow ((x^{\binom{r}{2}})^2 - 1) \bmod n = 0 \Rightarrow$$

$$((x^{\binom{r}{2}} - 1) * (x^{\binom{r}{2}} + 1)) \bmod n = 0$$

Therefore, unless $(x^{\binom{r}{2}}) = \pm 1 \bmod n$, at least one of the terms on the left side of the equation above must have a nontrivial factor in common with n . By finding

$\gcd(x^{\frac{r}{2}}-1, n)$ and $\gcd(x^{\frac{r}{2}}+1, n)$, we have a very good chance of finding a factor for n [19,20]. The problem is that finding 'r' on a classical computer is $O(n)$ and if we consider m as the number of the bits in n ($2^m = n$) then the problem of finding r becomes $O(2^m)$ which is exponential for growing m and therefore intractable. This is exactly where the idea of quantum parallelism from quantum computers comes to the aid. The details are explained in the next subsection.

2.2.2.1 How Shor's Algorithm Works

As it was explained above the main challenge is to find the period 'r' for the function $f_n(a)$. Shor proposes to do this using quantum parallelism offered by quantum computers. His algorithm first creates a quantum register consisting of two parts, namely register1 and register2. Next it loads register1 with the equally weighted superposition of numbers $a=0,1,2,\dots,q$ where $n^2 \leq q \leq 2n^2$ [19, 20].

Then it calculates $f_n(a)$ for all 'a' in quantum parallel and puts the result in register2. At this point the contents of registers 1 and 2 constitute a quantum system and their contents are related: register1 contains an equally weighted superposition of all values of 'a' from 0 to q and register2 contains a superposition of all values: $f_n(a)$. It is important at this point to remember that any measurement of either of the two registers will not only affect the contents of that register, but also it will affect the contents of the other register, as the contents of these two registers are related to each other (The contents of register2 were obtained by the application of the function $f_n()$ to the contents of register1).

Next a DFT is performed on the contents of register2 [20]. Since the contents of register2 were periodic before the application of DFT, after performing DFT, it will contain a superposition of the results of the DFT operation; *however*, this superposition is no longer an equally weighted one. The DFT of a pseudo-periodic sequence is very close to an impulse train, with impulses occurring at the multiples of the inverse period and the magnitude of the DFT sequence being very close to '0' anywhere else [20]. As a result if the contents of register2 and register1 are observed after performing the DFT on register2, with a very great likelihood the observed value in register1 will be a multiple of the period 'q/r'[19, 20]. To make this clearer, we remind the readers that Fourier Transform gives the frequency domain representation of the time domain signal. Therefore, if a continuous function is periodic in time domain its frequency domain representation will be a periodic train of impulses in which each impulse occurs at the multiples of the inverse of the period of the original time-domain function (i.e. if $x(t)$ has a period of T_0 then $X(f)$ will be a train of impulses occurring at $kf_0 = \frac{k}{T_0}$). Similarly, if $x[n]$ is a quasi periodic sequence with a period 'r' and duration of q, then $X[k]=\text{DFT}\{x[n]\}$, will be a sequence which will have 'r' peaks, and each two successive peaks will be q/r apart from each other. Therefore once we measure the value of register2 (after the DFT operation on register2), the value in register1 will collapse to a multiple of 'q/r' with a great probability. Now, if the process explained here (loading register1 with equally weighted superposition of values of 1 to q, applying the $f_n()$ to register1 and putting the results in register2 and then taking the DFT of register2) is repeated a few

times a number of different multiples of the period 'q/r' is obtained from which the actual value of 'r' could be easily guessed.

Figures 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 give the DFT sequence for the quasi-periodic sequences:

$$x_1[n] = [3 \ 7 \ 6 \ 3 \ 7 \ 6 \dots 3 \ 7 \ 6] \text{ (period=3, length of the sequence=81)}$$

$$x_2[n] = [3 \ 7 \ 6 \ 3 \ 7 \ 6 \dots 3 \ 7 \ 6] \text{ (period=3, length of the sequence=729)}$$

$$x_3[n] = [1 \ 12 \ 2 \ 7 \ 4 \ 1 \ 12 \ 2 \ 7 \ 4 \dots 1 \ 12 \ 2 \ 7 \ 4] \text{ (period=5, length of the sequence=25)}$$

$$x_4[n] = [1 \ 12 \ 2 \ 7 \ 4 \ 1 \ 12 \ 2 \ 7 \ 4 \dots 1 \ 12 \ 2 \ 7 \ 4] \text{ (period=5, length of the sequence=3125)}$$

$$x_5[n] = [1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \ 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \dots 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17]$$

$$\text{(period=10, length of the sequence=100)}$$

$$x_6[n] = [1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \ 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17 \dots 1 \ 3 \ 5 \ 8 \ 2 \ 4 \ 11 \ 6 \ 25 \ 17]$$

$$\text{(period=10, length of the sequence=10000)}$$

As it could be clearly seen, both $X_1[k]$ and $X_2[k]$ have 3 peaks, the only difference is that since $x_2[n]$ has a much bigger length than $x_1[n]$, it is much closer to a 'periodic' signal than $x_1[n]$, therefore the peaks in $X_2[k]$ are much narrower than the peaks in $X_1[k]$. Similarly $X_3[k]$ and $X_4[k]$ have both 5 peaks but $X_4[k]$'s peaks are much narrower than those of $X_3[k]$, and again $X_5[k]$ and $X_6[k]$ have both 10 peaks but $X_6[k]$'s peaks are much narrower.

Figure 2.1 DFT of $x_1[n] = [3 \ 7 \ 6 \ \dots \ 3 \ 7 \ 6]$ (length of the sequence=81)

Figure 2.2 DFT of $x_2[n] = [3 \ 7 \ 6 \ \dots \ 3 \ 7 \ 6]$ (length of the sequence=729)

Figure 2.3 DFT of $x_3[n] = [1\ 12\ 2\ 7\ 4\ \dots\ 1\ 12\ 2\ 7\ 4]$ (length of the sequence=25)

Figure 2.4 DFT of $x_4[n] = [1\ 12\ 2\ 7\ 4\ 1\ \dots\ 1\ 12\ 2\ 7\ 4]$ (length of the sequence=3125)

Figure 2.5 DFT of $x_5[n] = [1\ 3\ 5\ 8\ 2\ 4\ 11\ 6\ 25\ 17\ \dots\ 1\ 3\ 5\ 8\ 2\ 4\ 11\ 6\ 25\ 17]$ (length of the sequence=100)

Figure 2.6 DFT of $x_6[n] = [1\ 3\ 5\ 8\ 2\ 4\ 11\ 6\ 25\ 17\ \dots\ 1\ 3\ 5\ 8\ 2\ 4\ 11\ 6\ 25\ 17]$ (length of the sequence=10000)

CHAPTER 3

THE CONCEPT AND MOTIVATIONS FOR THE HARDWARE BASED TECHNIQUE

3.1 The Main Idea

The main idea behind the Integration Technique (IT) is that many problems could be solved in two phases:

Phase one: Partition the problem and apply operations to each part simultaneously.

Phase two: Integrate and collect the results from the first phase to get the complete solution.

By partitioning a problem into tasks which could be performed simultaneously (and therefore independent from each other), we make it possible to reduce the time necessary to solve the problem by a great amount. This is achieved by sacrificing silicon area; as such the maximum size of the problems which could be tackled is limited by the maximum number of transistors per unit area of the silicon and the maximum size of the die. Therefore with each passing year bigger and bigger sizes of a problem could be solved. As a result the solution to these problems becomes fully scaleable with technology meaning that as technology advances both the increase of the maximum clock speed and the increase in the die size/transistor packing density will be fully utilized. To

have a better understanding of the concept of the IT some examples are given in the following subsections.

3.2 Logic Function of m Inputs

To understand the main 'concept' of IT the simplest of examples would be perhaps the implementation of logic functions. Any function $f_m(I_0, I_1, \dots, I_{m-1})$ could be implemented as a SUM of PRODUCTS. Therefore, the first phase would be to apply k number of AND operations to all 'm' inputs in parallel. Here k is the number of '1's in the truth table value for f_m . The second phase would be to integrate the results of the AND operations with a single OR operation which would combine the result of all AND operations in phase one. Since in the truth table for an m-input logic function there are 2^m different combination of the 'm' inputs, therefore the OR operation in the second phase will have 'k' inputs with $k \leq 2^m$. A k-input OR gate could be implemented as a balanced binary tree of 2-input OR gates (assuming that k is a power of 2). Such a tree would have 'k' leaves and therefore its number of levels would be $\log(k)$. This in turn would mean that the delay of such a tree would be: $(\log(k) * \text{delay of a 2-input OR gate})$. Figure 3.1 shows the detail of the two phased system mentioned above for a logic function with 4 inputs and assuming that $k=2^3$. Therefore the over all delay would be: $(\text{delay of the m-input AND gate}) + (m * \text{delay of a 2-input OR gate})$. In a uniprocessor system this delay would be equal to $(\text{delay of a k-input OR operation} + k * (\text{delay of an m-input AND operation}))$.

Figure 3.1: The IT applied to finding the value of a logic function with 4 inputs assuming that the truth table for the function gets a value of '1' only for 8 different combination of the 4 inputs. The 4-input AND gates in parallel, constitute the first phase of IT and the tree of the OR gates constitute the second phase.

3.3 Algebraic Functions

Any algebraic function $F(x)$ could be expressed in terms of the following series:

$$F(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

$F(x)$ could be obtained by applying different operations to x (in parallel) in the first phase and then adding the results by a multiple-input add operation in the second phase. Figure 3.2 illustrates a possible such scheme for an 8th degree polynomial. Note that in the first phase the powers of x (from 1 to 8) are calculated with as much parallelism as possible. Since generating powers of bigger than 1 for ' x ' is dependent on the previous generation of smaller powers, it is not possible to implement the first phase in complete parallelism. For example you can not generate x^3 , without having previously generated x^2 , it is however possible to generate $x^7 (= x^4 * x^3)$ and $x^5 (= x^4 * x)$ in parallel. Also notice that this whole function could have been implemented with one multiplier, one adder and a control circuitry. However, that would have made the calculation timing of a n -degree polynomial $O(n)$, whereas in the present situation our delay is in the order of $O(\log n)$. This significant speed up has been achieved at the cost of spending more silicon area: we are implementing IT with 15 multipliers and 8 adders. However, it should be mentioned here that this increase in the silicon area is actually what makes this scheme "scaleable" with technology: with each advancement in technology, more silicon will be available for our scheme and higher and higher degrees of polynomials could be calculated in logarithmic times by IT.

Figure 3.2 The Application of IT to the Problem of Calculating an 8th Degree Polynomial

3.4 Calculation of the Area Under a Curve $f(x)$

Another problem which could be tackled by IT is the calculation of the area under a curve $f(x)$. This area is actually the result of the following integration:

$$Area = \int_a^b f(x)dx$$

This integral in turn is numerically calculated as:

$$\int_a^b f(x)dx \cong \left(\frac{1}{2}Y_1 + Y_2 + Y_3 + \dots + Y_{N-1} + \frac{1}{2}Y_N \right) \Delta x, \text{ where } Y_1 = f(a) \text{ and } Y_N = f(b)$$

With the assumption that the values $x_1 \dots x_N$ are *simultaneously accessible*, IT could solve this problem in $O(\log N)$. Figure 3.3 shows the details of this scheme: First $x_1 \dots x_N$ are applied to N number of $f(\)$ logic blocks in *parallel*, to obtain $Y_1 \dots Y_N$, this constitutes the first phase of IT. Then, the values $Y_1 \dots Y_N$ are pair wise added (in parallel), then the results of these additions will be again pair wise (again in parallel) added and so on and so forth until the final value for the over-all sum has been calculated, then finally a multiplication with Δx will be performed. This calculation of the final sum from the pair wise addition of $Y_1 \dots Y_N$ (using a binary tree of full-adders) and the final multiplication of the result with Δx will constitute the second phase of the IT. Again as it could be seen a large amount of silicon has been used to achieve logarithmic timing. Again this means that with each new technology, more accurate integrations (smaller Δx s) could be achieved.

Figure 3.3 Using IT to Calculate $\int_a^b f(x) dx$

3.5 Vector Multiplication

The problem of vector multiplication is another area that could benefit from IT. The simplest example would be the calculation of $C_{1 \times 1} = A_{1 \times n} \times B_{n \times 1}$. This problem could be solved by our technique as follows:

Phase one: Each element in A is multiplied with its corresponding element in B in parallel (simultaneously), the delay for this phase will be the delay of one multiplication operation. The number of multiplier units necessary for this part would be n.

Phase two: Add the result of corresponding multiplications from the phase one with one block of adders, to get the values for c_{11} . Figure 3.4 shows the structure and the methodology that we just described for the one dimensional matrix multiplication.

As seen in the figure 3.4 the block of adders could be implemented as a binary tree of adders, with each adder taking two operands as its inputs. Therefore the over all delay for the second part will be $\log(n) \times \text{delay of one adder}$. The over all delay for the whole operation will be $(\text{delay of one multiplication}) + \log(n) \times (\text{delay of one addition})$. The delay of such an operation in a uniprocessor system would be:

$$\text{Delay of calculation of } \sum_{i=1}^n a_{1i} * b_{i1} = O(n).$$

Again this speed up has been achieved at the cost of extra silicon resources. As in the previous examples the scheme appears to be fully scaleable with technology, meaning that with each new generation of technology bigger and bigger sizes of matrices could be multiplied in this manner.

Figure 3.4 The IT applied to the problem of calculation of $C_{11} = A_{1 \times n} B_{n \times 1}$

CHAPTER 4

APPLICATION OF 'IT' TO THE FACTORING PROBLEM

4.1 Architectural Overview

As it was explained in chapter 2 the only part of Shor's algorithm which makes use of quantum computers is the part related to finding the period 'r' of the function $f_n(a) = x^a \bmod n$. The main idea behind the architecture proposed in this thesis is to use the concept of IT to find the period 'r' with a timing complexity of polynomial order of the $\log(n)$. In other words silicon area has been sacrificed in favour of reducing the timing complexity of finding 'r'. Once this period has been found, the same steps as Shor's algorithm are followed and the number n is factored. But before going into the explanation of how the proposed architecture works, it is very important to mention that based on the number theory theorems which could be found in any basic Number Theory books such as [21] the period 'r' (which is actually known in Number Theory as the order of $x \bmod n$) for an 'n' which is equal to the multiplication of two odd primes could be anywhere: $1 \leq r \leq (n/2)$.

The over-all architecture of the proposed system is based on generating the number

$f_n(a) = x^a \pmod n$ for different values of 'a' from 1 to $n/2$ and searching through the results to find the first value of 'a' which results in the number $f_n(a)=1$. $f_n(a)$ is obviously equal to '1' for $a=0$, however, after that it becomes equal to '1' only for those values of 'a' which are exact multiples of 'r' [21]. The problem is that generating all the powers of $x \pmod n$, from 1 to $n/2$ by conventional computers takes $O(n)$ time, and searching through the results to find the first occurrence of '1' takes another $O(n)$ time. Therefore we try an IT approach to reduce the complexity of this problem. This is done as follows:

- 1) Generate all the powers of $x \pmod n$ in which the exponent is a power of 2 followed by taking their modulo with respect to n , that is to say, generate:

$$x^a \pmod n \text{ where } a = 2^0 \dots 2^i \dots 2^{\lfloor \log(n/2) \rfloor}.$$

Let's call these generated powers as "basic powers". To get a better picture, if the exponent 'a' is written in binary radix, in the first step all basic powers of x :

$$x^1 \pmod n, x^{10} \pmod n, x^{100} \pmod n, x^{1000} \pmod n, \dots, x^{100\dots00} \pmod n$$

are calculated. It will then be possible to create all the remaining powers of x using the generated powers in this step. As it will be shown later on, the completion of this stage will take only $O(\log n)$ time.

- 2) Generate all the intermediate powers: $x^a \pmod n$ for 'a' from 1 to $n/2$. This could be done as follows: for each bit of the exponent which is a '1', the corresponding basic power of x generated in the first step will be multiplied. So, for example:

$$x^{101} \pmod n = [(x^{100} \pmod n) * x^1 \pmod n] \pmod n$$

$$\text{or } x^{1010} \pmod n = [x^{1000} \pmod n * x^{10} \pmod n] \pmod n$$

$$\text{or } x^{1110} \pmod{n} = (([x^{1000} \pmod{n} * x^{100} \pmod{n}] \pmod{n}) * x^{10} \pmod{n}) \pmod{n}$$

- 3) Once all of the powers of x modulo n have been generated, a “search” should be made to find the first power of x modulo n which generates a ‘1’. The search scheme used in this stage has to have a timing complexity in the logarithmic order. A search scheme with such timing complexity will be explained in the chapter 5.

Figure 4.1 shows the block diagram of a system to find the order of $x \bmod n$ for a 4-bit number ‘ n ’. Although this block diagram corresponds to a 4-bit system, it could be easily generalized to any m -bit system. In this example (figure 4.1), the number 4 which is co-prime to 15 is randomly selected. Finding a number co-prime to the number to be factored is a task which could be performed by the conventional computational methods in logarithmic time [20]. If the number to be factored is 15, then the order of $x \bmod 15$ could be anywhere between 1 and 7. The purpose as mentioned before is to generate all powers $x^a \bmod 15$ for ‘ a ’ between 1 and 7. As it could be seen in figure 4.1, in the first step (upper part) the basic powers ($x \bmod n$, $x^2 \bmod n$ and $x^4 \bmod n$) are generated, the time required for the completion of this part is $O(\log n)$. The second step (lower part) generates all the intermediate powers ($x^3 \bmod n$, $x^5 \bmod n$, $x^6 \bmod n$ and $x^7 \bmod n$), using the basic powers generated in the first step.

Figure 4.1 The Block Diagram for the over all System (4-Bit Wide), in This Example

$n=15$ and $x=4$

4.2 First Stage

In the first stage we try to generate the basic powers (powers of x , where the exponent is a power of 2 such as 2, 4, 8, 16, etc.). Since in order to generate any $x^{(2^k)}$ we need to already have $x^{(2^{(k-1)})}$, the process of generating the basic powers could not be parallelized. The implementation of the data path for the first step is shown in figure 4.2. Since the period ' r ' could be anywhere between 1 and $n/2$, we will have to generate ' $m-1$ ' number of powers of x modulo n , where $m = \text{ceil}(\log(n))$. Each of these powers of $x \pmod{n}$ will reside in one register. At each rising edge of the clock, t_i , R_i is enabled and the output of the R_{i-1} is selected. The selected output will get squared and then divided by n and the remainder of the division is written into the enabled register (i.e. $R_i = (R_{i-1} * R_{i-1}) \pmod{n}$). In the first clock cycle when the "start" signal is received, no output from any register is selected, instead R_0 is enabled, and since its input is connected to x , it gets written with the value x . But from that clock cycle on at each clock cycle i the i^{th} register is enabled and the output of the $(i-1)^{\text{th}}$ register is selected to be squared/divided and the remainder to be written in R_i . This process has to go on until all the powers of x with the exponents of 2^i ($(0 \leq i \leq \text{ceil}(\log(n/2)))$) have been generated. The signals "ith_power_generated" (for $0 \leq i \leq \text{ceil}(\log(n/2))$) are generated by the controller. Each ith_power_generated signal is asserted by the controller at the rising edge of clock $i+1$ and is kept asserted from that moment on, until the end of the algorithm. ith_power_generated signals signify the fact that the $x^{(2^i)} \pmod{n}$ has been generated

and has been stored in the register R_i . The process of generating the “basic powers”

will take a time:

$$t_{up} = O((i_{max} + 1) * \text{clock_cycle}) = O(\text{ceil}(\log(n/2)) * \text{clock_cycle}) = O(\log(n) * \text{clock_cycle}).$$

Figure 4.2 Datapath for the First Stage (Upper Part)

What should however be realized is that if the hardware is designed with m -bit-wide units, and $m-1$ number of registers (i.e. the maximum number to be factored using such a system is at most $2^m - 1$) then unless precautions are taken even for an input with a bit number less than m (e.g. when we want to factor a number less than $2^{m/2}$) we will have to generate $m-1$ powers of x modulo n . This means that we are generating unnecessary powers of x . Therefore the following precaution is taken: a combinational circuit (left part of the figure 4.2) will take n as its input and based on the location of its most significant bit the “ended” is selected from among the i th_power_generated signals generated by the controller. The “ended” signal signifies to the lower part circuit that the generation of all necessary basic powers has finished and that the lower part circuit could start generating the intermediate powers. The following is the algorithm according to which the controller works:

0) Enable R_0 select no output from any register.

1) Enable R_1 . Select R_0 's output. Assert 0^{th} _power_generated signal.

2) Enable R_2 . Select R_1 's output. Assert 0^{th} _power_generated signal. Assert 1^{th} _power_generated signal.

....

i) Enable R_i . Select R_{i-1} 's output. Assert 0^{th} _power_generated signal. Assert 1^{th} _power_generated signal. Assert 2^{th} _power_generated signal.... Assert $i-1^{\text{th}}$ _power_generated signal.

....

m) Enable R_m . Select R_{m-1} 's output. Assert 0^{th} _power_generated signal. Assert 1^{th} _power_generated signal. Assert 2^{th} _power_generated signal.... Assert $i-1^{\text{th}}$ _power_generated signal.... Assert $(m-1)^{\text{th}}$ _power_generated signal
 $m+1$) Assert 0^{th} _power_generated signal. Assert 1^{th} _power_generated signal. Assert 2^{th} _power_generated signal....
 Assert $i-1^{\text{th}}$ _power_generated signal....
 Assert $(m-1)^{\text{th}}$ _power_generated signal. Assert m^{th} _power_generated signal.

The figure 4.3 shows the timing diagram for the outputs of the controller for the first stage. Although the signal “ended” is not a controller output (it is selected from i^{th} _power_generated signals in the datapath, see the left side of the figure 4.2), for clarity of the argument, and to see that signals relation to i^{th} _power_generated signals, the timing behaviour of the “ended” signal for various possible values of n has been shown. Also registers $R_0...R_6$ don't belong to the controller, however, again for the sake of clarity their values have been shown in the figure 4.3.

Figure 4.3 Timing diagram for the output signals of the upper part controller, registers $R_0 \dots R_6$ and the signal “ended” are not part of the controller, they belong to the datapath (figure 4.2), however for the sake of clarity they have been included in this figure.

4.3 Second Stage

In this step we try to generate all the remaining powers of x , using the basic powers generated in the stage 1 (the upper part, see figure 4.1) using parallel ALU units as shown

in figure 4.4. Note that in figure 4.4 each ALU unit is capable of multiplying its two inputs and then making the modulo operation of the multiplication result with 'n'. Please also note that '%' has been borrowed from the C language's syntax and signifies the modulo operation. Here is how we do it:

- i) $(X^1)\%n$: already available from stage 1.
 - ii) $(X^2)\%n$: already available from stage 1.
 - iii) $(X^3)\%n = (((X^1)\%n) * ((X^2)\%n)) \% n$ (both from stage 1)
 - iv) $(X^4)\%n$: already available from stage 1.
 - v) $(X^5)\%n = (((X^4)\%n) * ((X^1)\%n)) \% n$ (both from stage 1)
 - vi) $(X^6)\%n = (((X^4)\%n) * ((X^2)\%n)) \% n$ (both from stage 1)
 - vii) $(X^7)\%n = (((X^4)\%n) * ((X^3)\%n)) \% n$ (X^4 from stage 1, X^3 from iii)
 - viii) $(X^8)\%n$: already available from stage 1.
 - ix) $(X^9)\%n = (((X^8)\%n) * ((X^1)\%n)) \% n$ (both from stage 1)
- and so on and so forth....

Figure 4.4 shows the details of the lower part of a 5-bit system. Note that although, $(X^7)\%n$ is generated by the multiplication of $(X^1)\%n$, $(X^2)\%n$ and $(X^4)\%n$, since $(X^3)\%n$ has already been generated by the multiplication of $(X^1)\%n$ and $(X^2)\%n$, we can use the result of that multiplication and generate $(X^7)\%n$ by multiplying $(X^4)\%n$ and $(X^3)\%n$. This way we don't waste silicon area to generate $(X^3)\%n$ twice, once we have generated $(X^3)\%n$, we can use it to generate any higher power such as $(X^7)\%n$ or $(X^{11})\%n$ (in case of a 5-bit-wide system or larger).

Calculating the time required for calculating all these powers is a straight forward matter:

Any power of $x \pmod n$ is generated by the multiplying the basic powers corresponding to the bits in the exponent which are '1'. So to generate $x^a \pmod n$, if 'a' has 'L' number of '1's in its binary representation, we have to multiply 'L' number of basic powers. To multiply 'L' numbers to each other, we have to make 'L-1' number of multiplications, and we shouldn't forget that since we are doing 'modular' multiplication each of the 'L-1' multiplications is followed by a modulo operation. This means that to produce $x^a \pmod n$, if 'a' has 'L' number of '1's in its binary representation then the total delay to generate $x^a \pmod n$ will be:

$$t_a = \text{delay of upper part} + (L-1) * (\text{delay of multiplication unit} + \text{delay of the modulo unit})$$

$$(4.1)$$

So for example:

To produce $x^{15} \% n$, since 15 (which is $(1111)_2$) has four '1's in its binary representation. Then we need to make 3 multiplications (each multiplication is followed by a modulo operation):

$$x^{15} \% n = ((x^8 \% n) * ((x^4 \% n) * ((x^2 \% n) * (x^1 \% n)) \% n) \% n) \% n$$

And therefore: $t_{15} = \text{delay of the upper part} + 3 * (\text{delay of multiplication unit} + \text{delay of modulo unit})$

Since the exponent 'a' in $x^a \pmod n$ will be always less than $n/2$, the maximum number of '1's that 'a' could have in its binary representation will be: $\log(n)-1$, therefore the time delay of producing all the powers $x^a \pmod n$, for $1 \leq a \leq n/2$ is:

$$t_{dl} = [(\text{\#of bits in } n/2) - 1] * (\text{delay of multiplication unit} + \text{delay of the modulo unit}) +$$

delay of upper part $\Rightarrow$

$$t_{dl} = O(\log(n)) * (\text{delay of multiplication unit} + \text{delay of modulo unit} + \text{clock_cycle})$$

(4.2)

Here n is the maximum number to be factored using this circuit and therefore, $\log(n)$ is the width of the arithmetic units in the circuit.

Figure 4.4 The lower part of a 5-bit system. The critical path has been highlighted. Note that the critical path corresponds to the generation of $x^{15} \pmod n$, and it involves 3 multiplier units followed by modulo operation (in binary representation of 15 there are four '1's and $4-1=3$).

4.4 A Remark Regarding the Time Delay Formula

As it could be seen from the formula 4.2, the time required for all powers to be generated is a function one of whose variables is the clock period. On the other hand, as the width of the architecture grows (larger and larger numbers are to be factored), the width of all arithmetic units (multipliers, dividers for modulo operation, adders, subtractors, etc.) grow (i.e. the delay in their critical paths grows), also as the width of the architecture grows, so does the number of states in the state machines and this reduces the maximum clock frequency of the system. Therefore the growth rate in the clock period as a function of the width of the architecture should be formulized, and if this growth rate is not of a polynomial order with the “width of the architecture” (width of the architecture = $O(\log(n))$), then the over-all system will not have a logarithmic timing complexity as it had been purposed. In Chapter 6, we will show that this growth rate could be indeed made logarithmic.

Another very important issue which must be understood is the following:

If we consider ‘s’ as an *arbitrary* number being factored by our architecture and consider ‘n’ as the *maximum* number which could be factored by our architecture, then the system as it has been described so far will generate all the powers $x^a \pmod s$ for ‘a’ between 1 and $n/2$, even if $s < n$, where n is the maximum modulo which could be handled by this circuit (width of the architecture is equal to $\log(n)$). So if the system has been designed with units of 8-bit width, the same amount of time will be required to generate the powers of $4 \pmod{15}$ as the time will be required for generating those of $17 \pmod{209}$. The reason

is that we are generating “unnecessary” powers for $4 \bmod 15$. In case of $4 \bmod 15$, we only have to generate the first 7 powers, where as in case of $17 \bmod 209$ we have to generate 127 powers. This is because the period for $4^a \bmod 15$ can not be bigger than 7 and the period for $17^a \bmod 209$ can not be bigger than 127. In general since we know that for $x \bmod s$, the period we look for, is at most $s/2$, the search will be limited to powers from 1 to $s/2$ and *NOT* from 1 to $n/2$. Therefore if the order of $x \bmod s$ is to be found, only generation of powers from 1 to $s/2$ will be necessary and generation of any higher power will be unnecessary. This is the subject which was also mentioned in section 4.2. On the top left part of the figure 4.2. To avoid generating unnecessary powers the following precaution is taken: each multiply-modulo unit in the lower part generates and “ended” signal of its own and the combination logic block, shown on the top left corner of figure 4.2, will be used in the lower part to select one of the “ended” signals, based on the position of the most significant bit in the modulo number. The selected “ended” signal will be sent to the search circuitry (the third stage) as a “start” signal so that the third stage would know that all the “necessary powers” have been generated and it could start making its search among them.

4.5 Third Stage

We will not go into details of this stage in this chapter, as the whole of chapter 5 has been dedicated to the explanation of a new hardware based ‘search’ scheme. This ‘search’

scheme is what we use in the third stage. Here in this chapter it suffices that we only make the following remarks:

- At the end of the second stage we will have all powers of $x \bmod n$, from 1 to $n/2$ ready and available. Since each power is accessible *independently* from the others, all these powers could be accessed simultaneously and in parallel (See figure 4.4).
- Our task is to make a 'search' between all these generated powers and find the first occurrence of the value '1'. However, we have to make this 'search' in logarithmic time (with respect to the number 'n').
- We should keep in mind that the generated powers are *not ordered*.

CHAPTER 5

A NEW CIRCUIT TO REDUCE THE COMPLEXITY OF SEARCHING

5.1 Complexity of Search Algorithms in General

The search problem could be defined as: given a collection of numbers $\{J_0 \dots J_{n-1}\}$ and a key number 'k', find if any number the set $\{J_0 \dots J_{n-1}\}$ is equal to 'k' and if some numbers in the set are equal to 'k' then give the index of the first element of the set which is equal to 'k'. In general, computing systems are such that they could access the data in memory sequentially (one at a time). As a result, algorithms designed for conventional computing systems assume a sequential data access. Therefore, unless a binary search tree data structure is involved or that the search is performed on a sorted list of numbers, the complexity of a search algorithm is of the order of $O(n)$ [22]. If a binary search is performed its complexity will be $O(\log(n))$, but that would require an already sorted array of numbers [22]. The complexity of search using a binary search tree is on average $O(\log(n))$ and in the worst case $O(n)$, depending on how well balanced the tree is [22]. The circuit presented in this chapter makes a search on an unsorted collection of numbers and comes up with the solution in $\log(n)$ steps. Each step is performed in one clock cycle and the minimum length of the clock cycle grows with logarithmic scale with respect to

‘n’. The fact that the circuit presented here could be designed in a way so that its minimum clock period grows logarithmically with the increasing number of items to be searched will be proved in the next chapter. The application of such a circuit would be in situations where all the numbers in the collection could be accessed simultaneously. One possible application of this circuit is presented in this chapter and it is to make a search between the numbers $f_n(a) = x^a \pmod n$, $a, x, n \in \mathcal{N}$, $1 \leq a \leq (n/2)$ and find the smallest value of a for which $f_n(a) = 1$. Indeed the circuit presented here is used as the “second phase” of the IT applied to the problem of factoring. In other words the circuit presented in this chapter constitutes the third stage (search block) in the figure 4.1.

5.2 The Proposed Search Algorithm, Implemented in Hardware

We need to have a search algorithm implemented in hardware which would find the *first* occurrence of a number in a set of given numbers, in logarithmic time. Before giving a formal description of how the search scheme works, let’s first give an example. Lets assume that we are searching for the number $k = '8'$ in a set of eight numbers: $J = \{9, 5, 3, 8, 8, 7, 8, 2\}$. The purpose is to find the first occurrence of ‘8’ from the right in this set (i.e. the second element). Figure 5.1 explains the details of the scheme. First all eight elements of the set are compared simultaneously with ‘8’. The result of these parallel comparisons will be a string of 8-bit length: $A1 = \{0, 0, 0, 1, 1, 0, 1, 0\}$. Now we will feed this string of bits to a tree of OR gates. The first layer of OR gates will have 4

Figure 5.1 An example of the search for the number '8' among a set of eight numbers:

{9,5,3,8,8,7,8,2}

gates and each OR gate will take two of the consecutive bits in the A1 as its inputs (see figure 5.1). The output of the first layer of OR gates will be a 4 bit sequence

$A2=\{0,1,1,1\}$. This sequence will be fed into the second layer of the OR gates. This second layer of the OR gates will have two OR gates, and the output of these two will be: $A3=\{1,1\}$. $A3$ will be fed into one last OR gate (the third layer of OR gates) and the output from this last OR gate will be '1'.

An FSM will traverse this binary tree of 7 OR gates, based on the value of their inputs. As we start from the root of the tree at each OR gate that we arrive we look at its two inputs, if the right input is a '1' we just go to the right sub-tree and if that input is a '0' then we check the left input, and if that input is a '1' then we go to the left sub-tree. If the output from the root of the tree (the single OR gate in the third layer of OR gates) is a '0', it means that the search has failed and the number was not found in the set.

Figure 5.2 gives the details of the search scheme for a general case of searching for a number 'k' in a set J of eight elements: $\{J_0...J_7\}$. As it could be seen in that figure, the result of the simultaneous comparison of elements in J with the number 'k' will be a bit string: $\{a_0...a_7\}$. This bit string is fed to the tree of OR gates.

Figure 5.3 shows the state transition diagram for the FSM which will traverse the tree of OR gates in figure 5.2 and find the first element in J (from the right) which matches 'k'. As it could be seen from figure 5.3, the state transition diagram for this FSM has the exact same topology as figure 5.2. Figure 5.3 has a balanced binary tree structure and there is a one-to-one correspondence between all its internal nodes and the nodes of the tree in the figure 5.2. Also the leaves of the tree in the figure 5.3 correspond to $\{a_0...a_7\}$ in figure 5.2. The edges of the tree in both figures 5.2 and 5.3 have the same

names to show their one-to-one correspondence. The edges are the set of e_{ij} 's where 'j' is the node from which e_{ij} begins from and 'i' is the node that e_{ij} ends at (for example $e_{3(14)}$)

Figure 5.2 An example of the simultaneous comparison of the key number to a set of numbers followed by the binary tree of 'OR' gates

Figure 5.3 The state transition diagram of the FSM to traverse the tree in figure 5.2

is the edge which starts from node 14 and ends at node 3 . Node 14 is a_7 which corresponds to state 14 and node 3 is the OR3 which corresponds to state 3). Figure 5.4 shows the general case of searching among 'n' numbers (the elements of the set $J = \{J_0, \dots, J_{(n-1)}\}$) to find a key number 'k'. The simultaneous comparison part is not shown in figure 5.4. The bit string $\{a_0, \dots, a_{(n-1)}\}$ is the result of the simultaneous comparison of the number 'k' with the elements in the set J. The bit string $\{a_0, \dots, a_{(n-1)}\}$ is fed to a balanced binary tree of OR gates, therefore $\{a_0, \dots, a_{(n-1)}\}$ will act as the leaves of

the tree (the lowest level of the tree). The first level of the OR gates will contain $n/2$ number of 2-input OR gates, the second layer of the OR gates will contain $n/2$ number of

Figure 5.4 The ‘OR’ gate tree to be traversed for a general case of searching among ‘n’ numbers

2-input OR gates and so on and so forth until we reach the root of the tree. The formal algorithm for the FSM which would traverse this balanced binary tree (and find the position of the first match with ‘k’) is as follows:

1) *Start at the root of the tree;*

```

2) if (out put from the right sub-tree == 1){
    search the right sub-tree;}

else{
    search the left sub-tree;}

3) if (we have reached a leaf) {
    output the index of the leaf;
    STOP;}

else {
go to step 2; }

```

The binary tree in figure 5.4 has n leaves, and therefore, starting from its root and traversing it according to the algorithm given above until we reach a leaf will take $O(\log(n))$ steps. Therefore the over-all timing for performing the search using this scheme will be in the order of $O(\log(n) * \text{clock cycle})$

5.3 An Important Note Regarding the Timing of this Search Scheme

Before getting into the subject matter of this section, the reader should realize that here in this section, we have used the parameter 's' again. The reason is exactly the same as why we used the parameter 's' in 4.4: As it will be shown in chapter 6, the minimum clock period grows as a function of the width of the architecture and width of the architecture is the direct determinant of the *maximum* number which could be factored using this

architecture. Therefore here in this section to distinguish between an *arbitrary* number which is to be factored using our architecture and the *maximum* number to be factored using our scheme, we have used the parameter 's' to signify any arbitrary number which is to be factored with our scheme and the parameter 'n' is used to signify the *maximum number which could be factored using our scheme*.

As it was mentioned earlier in 4.2 and again in 4.4, if we try to find the order of $x \bmod s$ with a system with a width 'm' such that $m > \text{ceil}(\log(s))$, then we don't have to generate all the $2^{m-1} - 1$ powers. If this is the case then, since the number of generated powers will be $2^{s-1} - 1$, the search will be among a much fewer numbers. Therefore we don't have to start our traversal of the OR tree (in figure 5.1) from its root. For example for finding the 'r' for $x^a \bmod 15$ in an 8-bit architecture, we should realize that r can not be bigger than 7 and therefore, it is enough to start the search from the 3rd level of the tree from the bottom on the right most position and traverse the tree from there. Or if we generalize this assertion then for finding the 'r_s' for a number $s < n$, since $r_s < s/2$, it is enough to start traversing the tree from the right most position on the level: $\text{ceil}(\log(s/2))$ of the tree. Therefore the combinational logic block mentioned in section 4.2 and 4.4 (see the top left corner of figure 4.2), which recognizes the position of the most significant '1' of the modulo number 's' (in the figure 4.2 the modulo number is shown as 'n' and not 's'), can let the FSM in the search algorithm know from which position in the search tree it should start the process of traversing. Therefore in general the over all delay of the circuit to find r_s for a number s will be:

$t = \text{upper part delay} + \text{lower part delay} + \text{search delay} = O(\log(s) * \text{clock cycle period} + \log(s) * (\text{delay of multiplier} + \text{delay of modulo unit}) + \log(s) * \text{clock cycle period}).$

(5.1)

5.4 A Final Remark

As a final remark in this chapter we mention again that, in this chapter we have introduced a search scheme implemented in hardware which searches in an unordered set of numbers for a key number and finds the first occurrence of that key number in that set (if in fact the key does match any member of the set). This search scheme does its search in a logarithmic number of steps with respect to the number of the members in the set.

The search scheme introduced in this chapter is used as the third step of chapter 4, to find the first occurrence of '1' in the set of powers of $x \pmod{s}$. In other words, as also explained in section 4.5, the circuit explained in this chapter, takes the set of powers:

$$x^a \pmod{s} \text{ for } a=1,2,\dots,s/2$$

from the step 2 in chapter 4 and finds the smallest value of 'a' for which $x^a \pmod{s} = 1$.

This value is equal to the period of the function $x^a \pmod{s}$ and is called the order of $x \pmod{s}$.

CHAPTER 6

THE RATE OF GROWTH IN THE MINIMUM CLOCK PERIOD

6.1 Importance of the Issue

As the bit number in the number to be factored increases, the width of the smallest architecture to factor that number grows too. For example an 8-bit number requires a system architecture in which ALU units are at least 8-bit wide. Also the FSM's which control the data-paths (i.e the controller for the upper part, the controllers for the dividers in the lower part and the controller which traverses the tree of OR gates in the third step) will have greater numbers of states. As the width of the ALU units and the number of states in the FSM's increase, the minimum clock period at which the system could operate will increase. For example, an 4-bit system can operate at much higher clock frequencies than an 8-bit system. Therefore as mentioned before in chapters 4 and 5 (4.4 and 5.1), we must show that the rate of growth in the minimum clock period as the width of the architecture grows could be easily made polynomial (polynomial with respect to the growing architecture 'width' is logarithmic with respect to the maximum number to be factored), otherwise the arithmetic units and the finite state machines (FSM) in the

circuit will be so slow that it may cause our factoring scheme to become polynomial with respect to the number to be factored (as opposed to a logarithmic complexity). Here in this chapter we will make an argument which will show that the rate of growth in minimum clock period of the whole system could be made of a polynomial order with respect to the growing 'width' of the architecture.

6.2 Delay of a Combinational Logic Unit in General

In this section we try to show the 'existence' of a 'very simple' way to implement any combinational logic function with the timing complexity of $O(m)$, where 'm' is the number of the inputs to the logic function. The argument is that if such a straight forward way of implementation without any sort of logic optimization exists, then any synthesis tool should be able to implement a described function with a timing of $O(m)$ or even better (considering that they do logic optimization). In fact in the section 6.5, part b (Maximum clock frequency of the search algorithm's controller), a combinational logic function will be discussed which has $O(m+2^m)$ inputs and still the over-all delay is $O(m)$. Therefore it should not be thought that these implementation methods are the actual methods used in our system. These methods are just shown as a proof of concept that any modern synthesis tool should be able to come up with an implementation of the same or better timing complexity.

In general any combinational function could be represented as a tree of multiplexers (see figure 6.1). The number of levels in such a tree will be equal to the number of the inputs

to that combinational logic function: $I_1, I_2, \dots, I_m$. Therefore it is easy to see that in general a combinational logic *could* be implemented with a timing delay complexity of $O(m)$ where m is the number of the inputs to it. The tree of multiplexers in figure 6.1, is logically equivalent to a 2^m to 1 multiplexer and in fact many FPGA's base their programmable logic elements on multiplexers (by setting the inputs to the multiplexer as the value of the truth table of the function, and the inputs to the function activating the select signals of the multiplexer, very much the same as figure 6.1).

The fact that any combinational function could be implemented with $O(m)$ timing delay, (where m is the number of inputs to the combinational function) will make the task of determining the timing delay of the ALU units, the maximum clock frequency of the search state machine and the maximum clock frequency of the over all system very straight forward.

6.3 Timing Delay of a Multiplier of the Size m by m

As it could be found in any basic book about digital systems and/or computer organization such as [23] the delay of a combinational multiplier array of m bits by m bits is $O(2m)=O(m)$. There are also a variety of other schemes to implement modular multiplication, the commonest of which is the Montgomery's algorithm. However, since schemes based on Montgomery's algorithm give the result of $Z=X*Y*2^{-m} \pmod{M}$, where M is the m -bit modulus, they wont provide our system with much benefit, unless an extra multiplication with $2^m \pmod{M}$ is performed at the end [24, 25]. But even then,

the Montgomery algorithm will do nothing special in our case that a regular multiplier implemented as an algorithmic state machine which successively shifts and adds the multiplicand wouldn't do.

Figure 6.1 The tree of multiplexers to implement any function F . The P_i 's are equal to 1 or 0, depending on the value of F at that particular combination of $I_1 \dots I_m$.

Also Montgomery's algorithm is useful specially when there is pipelining involved and our system is not a pipelined one. However, as mentioned earlier, it is possible to implement the multiplier as an algorithmic state machine which is based on the successive left shifting and adding the multiplicand based on the bit values of the multiplier and this would reduce the area considerably (compared to the combinational multiplier). A multiplier implemented as an algorithmic state machine (mentioned above) could find the result of the multiplication in $O(m)$ clock cycles [26]; however, it would add to the over all time required for doing the multiplication (compared to the combinational multiplier which is clock independent). In the implementation of the system proposed in this paper, we used a combinational multiplier.

6.4 Timing Complexity of a Divider of the Size $2m$ -bit by m -bit

There are a number of schemes to implement the divider circuit, some of which could be found in [26, 27]. Here in this chapter we will analyze two common schemes. The first is the one which implements the divider as a combinational circuit and the second is the one which implements the divider as an algorithmic state machine which finds the quotient and the remainder by successively subtracting the denominator from the numerator. Of the two, the former will be explained here and the latter will be explained in details in the next sub-section, in the context of maximum clock frequency of a state machine.

As it was explained before it is possible to implement a combinational function with n inputs as a unit with a timing delay of $O(n)$. Since in a $2m$ -bit/ m -bit divider there are

altogether $3m$ inputs the order of the complexity for the timing delay will be $O(3m)$ which is the same as $O(m)$, where in this case m is the number of bits in the maximum number to be factored. Therefore it is logarithmic with the size of the maximum number to be factored using this scheme

6.5 Maximum Clock Frequency of a FSM

In our circuit, apart from the search algorithm which has been implemented using a FSM, dividers of the lower part have also been implemented as a FSM controlling a divider data path. It is therefore appropriate if we first analyze the rate of growth in the minimum clock period in a FSM. In general FSMs could be represented as a state register, which holds the bits of the current state, and a combinational unit which decodes the next state and the outputs based on current state and the inputs to the FSM (see figure 6.2).

The main determinant of the maximum clock frequency of the FSM, is the delay of the next state decoding unit. This unit in turn is nothing but an ordinary combinational logic, with h inputs from the current state and k inputs from the inputs coming from outside. Therefore, based on the argument in 6.2 we know that we can implement the next state decoding unit with a delay in the order of $O(h+k)$. In a FSM with n states, $h=\log(n)$ (unless some different state encoding method such as one-hot-encoding etc. is chosen which is not the case for us). If we consider our own case where the inputs to the FSM from outside are pretty much fixed to signals such as reset, rst and start then it is easy to see that as the maximum number to be factored grows the only thing which grows is the

number of the states and not the inputs from outside to FSM. Therefore the delay of the next state decoding unit is of order: $O(h)=O(\log(n))$, where n is the maximum number which could be factored using this circuit.

Figure 6.2 A model for Finite State Machine (FSM)

Now let's have a look at different state machines implemented in our circuit:

a) Divider's controller unit: The divider has to shift the numerator bit by bit to the left and each time subtract the denominator from the left shifted portion of the numerator (see figure 6.3). The division algorithm is as follows:

1) $D := \text{denominator};$

2) $N := \text{numerator};$

3) $Q := 0;$ //reset the quotient to 0;

4) $R := 0;$ //reset the remainder 0;

5) if $N=0$, then:

 STOP;

6) if $R < D$ then:

 left shift both R and N, a '0' is shifted into N from the right and the most significant bit of N gets shifted into R from the right

else:

$R = R - D;$ if ($R \neq 0$)

$Q = Q + 1;$

7) Go to step 5;

The main components of the divider data path for our system are shown in figure 6.3. The width of the numerator is twice the width of the architecture (width of the multiplier units) therefore it is equal to $2m = 2 * \log(n)$, where n is the maximum number to be factored and m is the width of the architecture. The number of bits in the denominator is the same as the width of the architecture and therefore it is equal to $m = \log(n)$. The number of states of the divider controller therefore is of the order $O(m) = O(\log(n))$ which

means that the maximum delay of the next state decode unit for divider's controller is of the order $O(\log(m))=O(\log(\log(n)))$. It should also be realized that the critical path in the divider data path is from output of the register on the far left of figure 6.3, through the subtracting unit and the multiplexer and back to the same register's input on the far left of figure 6.3. The timing delay of this path is in the same order as the timing delay of the subtracting unit. Assuming a ripple carry adder structure, the delay of this unit will be at worst $O(m)$ where again m is the width of the arithmetic units in the system (i.e. multipliers and subtractors).

Figure 6.3 Main components of the divider data path

Therefore the delay timing complexity of this unit is of the order: $O(m)=O(\log(n))$. As a result it is easy to see that as the width of the divider unit grows its clock period will grow with a linear complexity of the width of the divider (i.e. $\log(n)$). Also the number of steps involved in the division algorithm given here is $O(2m)=O(m)$, where $2m$ is the number of bits in the numerator register. Therefore both the clock period and the number of steps involved in the division operation grow linearly with the width of the architecture.

b) Maximum clock frequency of the search algorithm's FSM: The maximum clock frequency of the search algorithm is a bit trickier to analyze than that of the other state machines in this system. As it was explained earlier in this section, in general the rate of growth of the minimum clock period could be made in the linear order of the total number of inputs to next state decoding logic. Ordinarily, the number of external inputs to the FSM is limited to a few reset, and start signals, meaning that as the number of states in the FSM grow, the number of external inputs doesn't increase. Therefore we argued that the delay in the next state decoding logic is mainly a function of the number of bits in the state register (which is the logarithm of the number of states). However, in case of the FSM for the search algorithm, the number of external inputs to the next state decoding logic is not limited to a few start and reset signals, to the contrary, this number grows linearly with the growing number of states in the FSM. Let's first find the total number of inputs to the next state decoding machine of this FSM. As mentioned in chapter 5 this FSM is to traverse a balanced binary tree with 2^{m-1} leaves, where m is the width of the architecture (i.e. the width of the ALU units in the system). This means that there will be $O(2^m)$ number of states in the FSM. Also all the edges of this balanced binary tree are fed into the FSM as "external inputs". The edges of the balanced binary

tree of OR gates shown in figure 5.1, are the “inputs” to the OR gates in the tree. Therefore there will be $O(2^m)$ number of edges in the balanced binary tree in figure 5.1 and the number of state bits in the state register will be: $O(m)$. Therefore the overall number of inputs (number of external inputs + the number of state bits in the state register) to the next state decoding combinational logic for this state machine is $O(m+2^m)$. This means that the delay of the next state decoding logic is $O(m+2^m)$, meaning that the growth rate of the minimum clock period will be exponential with the width of the architecture and not linear. However, it should be remembered that the argument given in 6.2 was to show the ‘existence’ of a method of implementation and used no logic optimization. In reality logic optimization will reduce the total amount of timing delay considerably. Therefore the timing estimate given in 6.2 could be in certain cases an over conservative estimate. We can show that in this particular case the minimum clock period of the FSM will grow linearly with respect to ‘m’. To see this we will suggest a simple modification to the next state decoding unit to show that it is possible to implement it with a timing of $O(m)$. As it could be seen from the algorithm given in section 5.2, at each state, the controller has to make a decision on which direction to take (i.e. whether to traverse the left sub-tree or the right sub-tree). For the “OR” gate corresponding to each state, if the input from the right sub-tree is a ‘1’, the FSM traverses the right sub-tree, and if that input is a ‘0’ and instead the input from the left sub-tree is a ‘1’ then the FSM will traverse the left sub-tree. Therefore, to make its decision, at each state, the FSM needs only two external inputs. Therefore, we don’t feed all the inputs to/outputs from the OR gates being input directly to the next state decoding unit. Instead, based on the state we are in, we only select two inputs to the one OR gate

which corresponds to our state and feed those two inputs into the next state decoding logic. Based on the value of the inputs to this OR gate we make a decision to search whether the right or left sub-tree. Therefore the next state decoding logic will have $(\text{\#state bits in the state register} + 2)$ number of inputs and therefore it could easily be implemented with a timing complexity of $O(m)$. There remains the time required to select the two inputs from the all the available inputs to the OR gates in the tree. This selection is again done based on the current state we are in. The process of search finishes when we arrive at a leaf, therefore, decision making (as to whether to search the left or right sub-trees) is required only until we reach the lowest level of the tree. This means that from m number of state bits, only the $m-1$ least significant ones (or the most significant ones depending on the order in which we may encode each state) should be used to select which pair of inputs to OR gates to make decision. Figure 6.4 shows the modified model for this particular state machine.

Figure 6.4 The block diagram for the FSM in the search circuit

The block at the bottom of figure 6.4 which selects two signals from all input signals to the OR gates in figure 5.1, is logically equivalent to a single 2^{m-1} to 1 multiplexer,

however, in figure 6.4 we have labeled this block as a “tree of 2 to 1 multiplexers” and in figure 6.5 we have shown this block as a tree of 2 to 1 multiplexers with “m-1 select signals” and an over-all 2^{m-1} number of inputs. The reason for showing this block as a “tree of 2 to 1 multiplexers” rather than showing it as a single 2^{m-1} to 1 multiplexer, is to show that the delay of this block increases with a linear order with respect to the increasing ‘m’ (i.e. it is $O(m)$). A simple scheme with only 4 state-bits has been shown in figure 6.6 to clarify the situation, and figure 6.7 shows the tree of OR gates corresponding to this case. In figure 6.6 if all the three least significant bits are ‘1’s this would correspond to a leaf in the tree and therefore no decision making (in regards to the direction to take) is necessary (search stops). Based on the above argument, it becomes evident that the minimum clock period of such a search algorithms state machine would be $O(m)$ where m is the width of the multiplier units (i.e. the width of the architecture).

Figure 6.5: Details of the tree of 2 to 1 multiplexers (the block at the bottom of figure 6.4) which is logically equivalent to a single 2^{m-1} to 1 multiplexer. The delay of such a block would increase in the linear order with respect to the increasing 'm'

Figure 6.6 The tree of multiplexers for a search FSM with 4 state bits. We select the pair of inputs to the particular OR gate corresponding to the current state. Based on These inputs we make a decision as to go to left or right

Figure 6.7 The tree of OR gates to be traversed for a case with 4 state bits

c) Maximum clock frequency of the upper parts controller: An argument very similar to that in part a can be made for the controller of the upper part. The number of states in this state machine is in the order of $O(\log(n))$, where n is the maximum number to be factored using this circuit. The reason is that the number of states in this state machine linearly increases with the number of the “basic powers” that we try to generate in the upper part. These powers, as mentioned before, are $x^1 \pmod n, x^2 \pmod n, \dots, x^{2^{\log(n)}} \pmod n$, therefore their number is the same as the width of the architecture ($m=\log(n)$). Since the number of the states of this state machine is $O(\log(n))$, the number of bits used to encode its states will be $O(\log(\log(n)))$. Therefore the delay of the next state decoding unit is in the order of $O(\log(\log(n)))$. *However*, it is of great importance to realize that this is not the main bottle neck to determine the maximum operating clock frequency of the upper part.

The delay of the multiplier unit followed by the delay of the divider unit in the upper part, as again explained before, is $O(\log(n))$, which is far bigger than the order of complexity

of the next state decoding unit (i.e. $O(\log(\log(n)))$). Therefore, the main determinant of the clock frequency of the upper part is the delay of multiplication-modulo unit and is of the order of $O(\log(n))$.

The rate of growth of the minimum clock period of the over all circuit is in the order of:

$$\text{Max}\{\text{order of the upper part clock period, order of the lower part clock period, order of the search FSM clock period}\} = \max\{O(\log(n)), O(\log(n)), O(\log(n))\}$$

Therefore it could be seen that:

$$\text{Rate of growth of the minimum clock period for the over all circuit} = O(\log(n)).$$

(6.1)

6.6 The Over-all Timing Complexity of the Circuit

As it was mentioned in 4.1.4, the over all timing complexity of this circuit could be given as:

$$t = \text{upper part delay} + \text{lower part delay} + \text{search delay} = O(\log(s)) * \text{clock cycle period} + \log(s) * (\text{delay of multiplier} + \text{delay of modulo unit}) + \log(s) * \text{clock cycle period}.$$

Where, s is the number to be factored using this circuit. We have established in 6.3 and 6.4 that the timing delay of both the multiplier unit and the divider unit is $O(\log(n))$, therefore the timing delay of a multiplication followed by a division is $O(\log(n))$, also the complexity of the growth rate of the minimum clock period for the circuit is: $O(\log(n))$. Therefore we can substitute these values in the formula for over all timing complexity of the circuit and get:

$$t_{calc} = O(\log(s)) * \log(n) + \log(s) * \log(n) + \log(s) * \log(n) = O(\log(s) * \log(n)) \quad (6.2)$$

Where s is the number to be factored using this circuit and n is the maximum number which could be factored using this circuit.

CHAPTER 7

AREA COMPLEXITY OF THE CIRCUIT

7.1 The Order of the Rate of the Growth of Area

It is fairly obvious to see that the area of the over all circuit is directly related to the number of multipliers, the number of the dividers, the size of the search tree and the size of controllers. Additionally it should be noted that the area of each multiplier and each divider will grow as the maximum number to be factored using this circuit grows. The number of multipliers and dividers in the first stage of our system (upper part) is fixed (there is one multiplier and one divider). The number of multipliers and the number of the dividers in the lower part is:

$$\# \text{ of multipliers} = \# \text{ of dividers} = (n/2 - \# \text{ of basic powers}) = (n/2 - \log(n) + 1)$$

(7.1)

To see that the above formula is valid, it should be remembered that we are generating $n/2$ number of powers. The basic powers are generated in the upper part (the first stage of the system) and therefore the total number of powers generated in the lower part (the second stage of the system) is $(n/2 - \# \text{ of basic powers}) = (n/2 - \log(n) + 1)$. To generate each power, $x^a \pmod n$, we need to make $(L-1)$ number of multiplications where L is the

number of '1' bits in the binary representation of 'a'. However, for each power $x^a \pmod n$, we could see that:

$$a = 2^h + k$$

where $h+1$ is the position of the most significant bit of 'a' and 'k' is a binary number which is at most h-bit wide. This means that:

$$x^a \pmod n = (x^{2^h} \pmod n * x^k \pmod n) \pmod n$$

But since $k < a$, $x^k \pmod n$ is also one of the powers which is being generated, therefore, instead of generating the power $x^k \pmod n$ twice, we only generate it once and we can use it for generating any bigger power. Therefore generating $x^a \pmod n$ requires only one extra multiplication (followed by one division for the modulo operation), the rest of $(L-2)$ number of multiplications have already been made (while we were generating the power $x^k \pmod n$). Since each power requires one extra multiplication (followed by one division) the total number of multipliers (and also the total number of dividers) in the lower part will be as given in formula 7.1.

As for the area of each sub-unit:

a) The area of a multiplier unit: A multiplier unit in our circuit will grow with a complexity of $O(\log(n)^2)$ where n is the maximum number to be factored. This is because of the two-dimensional structure of the array of the combinational multiplication unit [23].

b) The area of a divider in the lower part: To analyze the growth rate of the area of the dividers in our circuit, we need to consider its major components:

* It is not so easy to make an estimate on the growth rate of area of the controller for the dividers in our system. This is mainly because the dividers in our circuit have controllers which have been described with behavioural code and how exactly the synthesis tool will synthesize and implement this behavioural code could not be known before hand. *However, if we assume* the approach shown in figure 6.1 for implementing a combinational function in general, it will follow that since the number of state bits is $O(\log(n))$, the number of inputs to the next state decoding unit will also be $O(\log(n))$ and this means that the number of multiplexers in the tree of multiplexer will be $O(n)$ and therefore the growth rate in the size of the next state decoding unit will be $O(n)$. The growth rate for the state register will also be $O(\log(n))$.

* Contrary to the controller unit for the dividers, making estimates for the growth rate of the area of the data path for the dividers is much easier (the data path is described using structural code). Subtracting unit (with a ripple carry adder) and the multiplexer unit in the divider data path both grow with $O(\# \text{ of bits in } n) = O(\log(n))$

c) *The area of the FSM for the search algorithm:* Again as mentioned earlier, this state machine has been designed using behavioural code and one can not know how the synthesis tool will exactly implement this behavioural code before hand. Therefore it is not easy to make an estimate regarding the growth rate of the controller for the search block ; *however, if we assume the models given in figures 6.4 and 6.5*, we can conclude that the area of the controller for the search algorithm grows with a complexity of $O(2^{\# \text{ of the state bits}}) = O(2^{\log(n)}) = O(n)$.

Also the area of the search OR tree will grow $O(n)$. These complexities show that the area of our system grows exponentially with the number of bits in n . As it will be seen the

next chapter under the heading of the experimental results: this fast growth rate, shows its significance as we try to implement the system in an FPGA. While an 8-bit system took only 30% of our FPGA resources, a 9-bit system could not fit on the FPGA.

CHAPTER 8

EXPERIMENTAL RESULTS

8.1 Methodology

Every circuit which we tested was implemented in an Altera Stratix EP1S40F780C5 FPGA. Table 8.1 gives the device features for Stratix EP1S40F780C5 [28].

Our methodology was as follows: Block by block all components in figure 4.1 were implemented (for an 8-bit system) and then tested. First the code for the upper part was written. After verification through simulation, this block was implemented in the FPGA and for a few pairs of x and n , the correct generation of basic powers was observed. Then the code for the lower part was written and after verification of the code through simulation, the combination of upper and lower parts was implemented in the FPGA. A piece of script in MATLAB was written, to find the order of $x \pmod n$, for a given pair of x and n . The powers generated by the combination of the upper part and lower part were compared to the values calculated by the MATLAB and the correct functionality of the implemented parts were verified. Then the code for the search circuit was written and again its correct functionality was verified through simulation. After this, the over all 8-bit system was implemented (the codes for all three parts were combined) in the FPGA and for a few pairs of x and n , the order of $x \pmod n$ was successfully found.

The growth in area was measured as follows: several architectures of increasing widths were developed and the amount of FPGA resources which had been consumed was recorded in each case, from this data the graph of Silicon resources (area) vs. Architecture width (in bits) was drawn. The exact numeric results for the area vs. the architecture width are given in the next section under the heading of “Results”.

In a similar way the growth rate in the minimum clock period was measured. Systems of varying widths were developed and for each architecture the clock rate for the simulation was increased until the ALTERA software (Quartus) issued warning(s) of timing violation on certain paths in the system. This way the maximum operational clock frequency (without any timing violation) for each system was recorded. The exact width of the architectures together with the maximum operational clock frequencies have been summarized in the next section.

Feature	
LEs (Logic Elements)	41,250
M512 RAM blocks (32 X 18 bits)	384
M4K RAM blocks (128 X 36 bits)	183
M-RAM blocks (4K X 144 bits)	4
Total RAM bits	3,423,744
DSP blocks	14
Embedded multipliers	112
PLLs	12
Maximum user I/O pins	822

Table 8.1 Device features for Stratix EP1S40F780C5

8.2 Results

All simulations that we present here were performed at gate level and the timing parameters related to all delays (including delay parameters related to place and route) were included.

The search algorithm that we implemented took a string of 128 bits of variable pattern as its input and returned the position of the least significant '1' in the pattern. The input bit string to the search circuit is represented in the actual Verilog code by a bit vector named: "cmp_res". The "cmp_res" bit vector is what we have represented in figure 5.4 as $\{a_0, \dots, a_{(n-1)}\}$. The comparators which would compare the searched key against the set of numbers (128 numbers in this test) were not included in the simulations for the functionality of the search circuit. Search circuit would get the bit string which is the result of this parallel comparison. The over all circuit to generate the powers of x modulo n and find the order of x (mod n) was implemented and its logarithmic timing was verified.

Figure 8.1 shows how the search circuit finds the position of the least significant '1' in the pattern $2^{38} + 2^{41} + 2^{68}$. In this test vector (cmp_res = $2^{38} + 2^{41} + 2^{68}$), if we take the least significant bits position as position 0, then the most significant '1' is in the 68th position, and there are also two other '1' bits in the positions 41 and 38 respectively. We wanted to restrict the search only to the first 64 bits in the string therefore, in figure 8.1 the number 'n' is equal to 64 indicating to the search circuit that the position of the least significant '1' can not be after the 64th position. "r_out" is a signal whose value has a

one-to-one correspondence with the current state of the search FSM. To be precise $r_out = (state_number + 2) / 2$, therefore, by looking at the value of r_out we can understand the current state of the FSM for the search circuit. Since in our test ' $n=64$ ', r_out toggles 6 times ($\log(64)=6$) before getting to its final value (the search tree is traversed from the right most node on the level right before the root, rather than starting from the root itself). In figure 8.1, had we set the value of ' n ' to 32, since the least significant '1' bit in the cmp_res is in the 38th position, the search would have ended in the "fail" state. The signal "ended_finding" signifies the end of the search operation. The signal "failed" shows that the number being searched has not been found (i.e. there is no '1' in the " cmp_res " bit string). The signal " r_out " shows the position of the first occurrence of the bit '1' in the bit string " cmp_res " and the signal "start" is asserted to initiate traversing the tree of the OR gates (figure 5.1).

Figure 8.1: The search algorithm finds the position of the least significant '1' in the pattern, `amp_res =  $2^{38} + 2^{41} + 2^{68}$` . The number 'n' signifies to the search circuitry that the least significant '1' can not be to the left of the 64th position in the string

Figure 8.2 shows the timing for finding the order of 4 modulo n , for $n \in \{15, 21, 35, 65, 247\}$, in an 8-bit system ($f=19\text{MHz}$).

Figure 8.3 shows the timing for finding the order of 13 modulo n , for $n \in \{15, 21, 55, 77, 209\}$, in an 8-bit system. It could be clearly seen that the amount of time taken by the calculations increases linearly with the increasing number of bits in the ' n '.

In figures 8.2 and 8.3: "ended" is the signal generated in the left part of figure 4.2 (this signal is selected from the i th_power_generated signals depending on the position of the most significant '1' in ' n ') to let the lower part (the second stage) know that all the basic powers have been generated. The signal "ended_finding" is a signal generated by the search circuit which signifies the end of the whole process and a successful finding of ' r ', the "failed" signal is to signify a failure in finding the value of ' r ' from generated powers and should never become high if the circuit is working properly, "reset_in" and "rst" signals are the synchronous and asynchronous reset signals respectively and they are used to reset the multipliers and the dividers in the lower part (i.e. the second stage), and finally the "start_in" signal is to start the upper parts controller to flow through the states in its state diagram and generate the basic powers.

Figure 8.2: As the number of bits in n increases by one the amount taken for calculation of r increases by a roughly constant amount, confirming the logarithmic complexity ($f=19\text{MHz}$, 8-bit system)

Figure 8.3: The logarithmic rate of increase in the time required to calculate the order of $x \bmod n$ is further verified, by using pairs of x and n different from those used in figure 8.2, in figure 8.2 x was equal to 4, in this figure x is equal to 13

Figure 8.4 gives the values for the time needed (at a clock frequency of 19MHz) for the calculation of the order of 4 modulo n (referred to as the quantity 'r' in all graphs), with n taking the values: 15 (4-bit), 21 (5-bit), 35 (6-bit), 65 (7-bit) and 247 (8-bit). The results in figure 8.4 were obtained from an 8-bit system.

Figure 8.4: Time delay for the calculation of 'r' increases linearly with the number of bits in 'n'

Figures 8.5 and 8.6 are the zoomed in versions of figure 8.3 and they show the timing details of finding the order of 13 (mod 15) and the order of 13 (mod 209) respectively.

Figure 8.7 shows the exponential growth in the area as the width of the architecture grows and figure 8.8 shows the linear trend of increase of the minimum period of the system clock, as the width of the architecture grows (confirming a logarithmic growth with respect to the maximum modulo number 'n' to be handled using this circuit).

Figure 8.5: Timing details of calculating the period of $f_x(a) = x^a \pmod n$ for $n=15$, $x=13$ (x and r are shown in binary)

Figure 8.6: Timing details of calculating the period of $f_x(a) = x^a \pmod n$ for $n=209$, $x=13$ (x and r are shown in binary)

Figure 8.7: The growth of area as the system width increases from 4 to 8 bit

Figure 8.8: Minimum clock period (in ns) as the system width increases from 4 to 8 bit.

CHAPTER 9

CONCLUSIONS AND FUTURE WORKS

9.1 Conclusions

As evident from the experimental results the timing complexity of finding the period of $f_n(a) = x^a \pmod n$ has indeed been reduced from a linear function of n to a logarithmic function of n . However this has been achieved at the cost of increasing the area growth rate complexity to polynomial rates (with respect to the number 'n', not with respect to the number of bits in 'n'). While this shows that we are fully utilizing all the resources available on the FPGA to tackle bigger and bigger problems at logarithmic times we reach the end of our FPGA resources when we implement an 8-bit system.

Although 8-bit was the biggest width we could achieve on the FPGA available to us, it should be taken into consideration that Stratix EP1S40F780C5 is not the leading edge FPGA available from ALTERA (the latest FPGA products from ALTERA are Stratix II family). In fact Stratix EP1S40F780C5 is not even the largest FPGA in its own family, the Stratix EP1S80 has twice as much resources as EP1S40 [28]. As such we should be able to fit a wider version of this architecture in a leading edge FPGA. Furthermore the fast trend of increase in both die size and the transistor density will make it possible to create wider and wider versions of this architecture. It is predicted that Moore's law will

preserve its validity for two more decades [3]. If this exponential rate of increase in the number of available transistors is taken into consideration: it could be predicted that in 20 years, a 36-bit version of our system could be built.

Also it is worth mentioning that one of the major significances of Peter Shor's various algorithms for the quantum computers was that it showed how quantum computers could be *useful* in solving certain problems. In other words his work was considered as a motivation for the research and development of the quantum computers. The largest number which could be factored by the latest model of a quantum computer (developed by IBM) is 15 [29]. Here in this thesis we propose an architecture which has not been implemented in a leading edge FPGA (i.e. larger FPGAs with greater resources are readily available in the industry) and yet we have managed to factorize 8-bit numbers.

9.2 Future Works

A study which we plan to pursue in future is to apply the idea of IT, into other difficult problems and hence to seek various methods for converting "timing complexity" into "growth of silicon area complexity". Any and all of the problems mentioned in chapter 3 could be a possible field for future research. If a problem could be parallelized then by sacrificing the silicon resources one can apply operations into the parallelized problem simultaneously and hence gain considerable time (at the cost of more silicon area).

Another possible work is to implement the architecture mentioned in this paper to ASIC (rather than FPGA). Implementing the system in ASIC will result in the ability to

implement a larger system than the one we have implemented in the FPGA. Furthermore, as mentioned in the chapter 8, the highest speed we could achieve for an 8-bit system is 19MHz, implementing the circuit in ASIC will also improve this speed.

REFERENCES:

- [1] Gordon E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, Vol. 38, No. 8, April 1965 , pp. 114-117.
- [2] G. E. Moore, "Progress in Digital Integrated Electronics," Technical Digest of International Electron Devices Meeting, Dec. 1975, p. 11.
- [3] G. E. Moore, "No Exponential is Forever: But 'Forever' Can Be Delayed!" IEEE International Solid-State Conference, Vol. 1, 2003, pp 20-23.
- [4] X. Huang, et al. "Sub 50-nm FinFET: PMOS," Technical Digest of International Electron Devices Meeting, Dec. 1999, pp. 67-70.
- [5] R. Chau et al, "Advanced Depleted Substrate Transistors: Single-gate, Double-gate and Tri-gate," Extended Abstracts of the 2002 International Conference on Solid-State Devices and Materials, Sept. 2002, pp. 68-69.
- [6] R. Chau, et al, "A 50nm Depleted-Substrate CMOS Transistor (DST)," Technical Digest of International Electron Devices Meeting, Dec. 2001, pp. 621-624.
- [7] S. Thompson, et al, "A 90nm Logic Technology Featuring 50nm Strained Silicon Channel Transistors, 7 Layers of Cu Interconnects, Low k ILD, and 1 μ m² 6-T STAM Cell," Technical Digest of International Electron Devices Meeting, Dec., 2002, pp 61-64.
- [8] Krishna Kavi, James Browne, and Anand Tripathi, "Computer Systems Research the Pressure Is On," IEEE Computer, January 1999, pp 30-39.
- [9] Wulf, Wm. A and McKee, "Hitting the Memory Wall: Implication of the Obvious," ACM Computer Architecture News Vol. 23, No 1, March 1995, pp 20-24

- [10] Christoforos E. Kozyrakis, Stylianos Peeissakis, and David Patterson “Scaleable Processors in the Billion-Transistor Era: IRAM,” *IEEE Computer*, September 1997, pp.75-78
- [11] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal, “Baring it all to Software: Raw Machines,” *Computer*, Volume 30, Issue 9, Sept. 1997, pp. 86 – 93.
- [12] Chang, Andrew, Dally, William J., Keckler, Stephen W., Carter, Nicholas P., Lee, and Whay Sing, “The Effects of Explicitly Parallel Mechanisms on the Multi-ALU Processor Cluster Pipelin,” *International Conference on Computer Design*, Austin, TX, October 1998. pp 474-481.
- [13] A. K. Lenstra, H.W. Lenstra Jr., M.S. Manasse and J.M. Pollard “The number field sieve.” *Proc. 22nd STOC*, 1990, pp. 564-572.
- [14] D. Silverman, “The multiple polynomial quadratic sieve”, *Math. Comp.*, Vol. 84 (1987) pp. 327–339.
- [15] J. Gerver, "Factoring Large Numbers with a Quadratic Sieve" *Math. Comput.* 41, 1983, pp 287-294.
- [16] M.A. Morrison, J. Brillhart, “A method of factoring and the factorization of F_7 ,” *Math. Comp.*, v. 29, 1975, pp 183-205.
- [17] H. W. Lenstra, Jr., “Factoring integers with elliptic curves”, *Ann. of Math*, Vol. 126 (1987) pp. 649–673
- [18] A. K. Lenstra, H.W. Lenstra Jr., M.S. Manasse and J.M. Pollard “The number field sieve.” *Proc. 22nd STOC*, 1990, pp. 564-572.

- [19] Peter Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," Proceedings 35th Annual Symposium on Foundations of Computer Science (1994), pp 124 - 134
- [20] Williams, Colin P. and Clearwater, Scott H. "Explorations in Quantum Computing," New York: Springer-Verlag, 1998.
- [21] E. Grosswald, "Topics from the Theory of Numbers," New York: The MacMillan Company, 1966.
- [22] T. Cormen, C. Leiserson, R. Rivest, C. Stein, "Introduction to Algorithms," McGraw Hill 1990 (2nd edition, 2001).
- [23] C. Hamacher, Z. Vranesic, S. Zaky, "Computer Organization" New York: McGraw-Hill, 2002.
- [24] P.L. Montgomery, "Modular Multiplication without Trial Division". Math. Computation, Vol.44, 1985, pp.519-521.
- [25] Stephen E. Eldridge and Colin D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," IEEE Transactions on Computers, Vol. 42, No. 6, June 1993, pp 693 – 699.
- [26] S. Brown and Z. Vranesic, "Fundamentals of Digital Logic with VHDL Design," New York: McGraw-Hill, 2000.
- [27] John L. Hennessy and David A. Patterson, "Computer organization and design : the hardware/software interface," San Mateo, Calif. : Morgan Kaufmann, c1994.
- [28] Stratix Device Family Data Sheet,
http://www.altera.com/literature/hb/stx/stratix_section_1_vol_1.pdf
- [29] IBM's Test-Tube Quantum Computer Makes History,

http://domino.research.ibm.com/comm/pr.nsf/pages/news.20011219_quantum.html

⑤ BL-85-124