KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI, GHANA.

APPLICATION OF RESIDUE NUMBER SYSTEM TO SMITH - WATERMAN ALGORITHM

BY

BAAGYERE, YELLAKUOR EDWARD, BSc. COMPUTER SCIENCE(HONS)

MAY, 2011

APPLICATION OF RESIDUE NUMBER SYSTEM TO SMITH - WATERMAN

ALGORITHM

BY

BAAGYERE, YELLAKUOR EDWARD, BSC. COMPUTER SCIENCE (HONS)

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING, KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF

MASTER OF PHILOSOPHY IN COMPUTER ENGINEERING

COLLEGE OF ENGINEERING

MAY, 2011.

Declaration

I, EDWARD YELLAKUOR BAAGYERE, hereby declare that this thesis is my own work and to the best of my knowledge it contains no academic material previously published or written by another person or group of persons, nor work which to a substantial extent has been accepted for the award of any other degree or diploma at Kwame Nkrumah University of Science and Technology (KNUST) or elsewhere or published in any academic journal, except where due acknowledgments are made in the thesis.

I therefore declare that the intellectual content of this thesis is the product of my own work, under the keen supervision of Dr. K. O. Boateng, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

.....	.....	.....
Student Name & ID	Signature	Date

Certified by:

.....	.....	.....
Supervisor's Name	Signature	Date

Certified by:

.....	.....	.....
Name of Head of Department	Signature	Date

Dedication

This thesis is dedicated to GOD ALMIGHTY, to all the members of the

BAAGYERE family and to my lovely Wife, MARY NARTEY.

Acknowledgements

First of all, I would like to express my deepest gratitude to my Supervisor, Dr. K. O. Boateng, who as a great mentor had inspired me to be a better person both in life and engineering science. He has exposed me to the basics of computer engineering, critical analysis of scientific issues, and the art of neat technical writing. Dr. Boateng, I can hardly forget the six (6) hours we spent on the other day trying to get a design pattern for the RNS forward converter. What more can I say better than to ask for GOD's blessing and favor to be on you in all your endeavors.

Next, I especially would like to express millions of thanks and gratitude to Dr. K. A. Gbolagade, the Head of Department of Computer Science, University for Development Studies, Navrongo Campus, for the coaching and inspirations he had bestowed on my academic life. The concepts of "Residue Number System" (RNS) you brought on my way is bearing fruits. Bravo Dr.

I own a thank you and appreciation to the Head of Department of the Computer Engineering, KNUST, and the staff for all the administrative and academic support.

I'm very much grateful to Prof. dr. Sagary Nokoe, the immediate past Vice-Chancellor of the University for Development Studies (UDS), for his moral, administrative, and academic supports. He has been a great mentor in my academic life. I wish to also express my sincere appreciation to Dr. Elkanah Oyetunji, the Dean of the faculty of Mathematical Sciences, UDS, for his constant advice and encouragement to close my open questions. Dr. Oyetunji, I have closed all the open question now.

I'm thankful to Mr. and Mrs. Isaac Baagyere, Mr. and Mrs. Jonas Baagyere, Francisca Baagyere, the entire Baagyere sons and daughters, and the entire An-yir family for the dozen of love, support and encouragement throughout all the years. May the Good Lord give you long live to enjoy the fruits of your labor.

I would specially like to thank my dear wife, Mary Nartey, a lovely lady God brought alone my way to be eyes and help meet unto me in the "wilderness" of my life. Millions kudos "Paaristic".

Finally, I am thankful to these people, Evans Alhassan, Patience Alhassan, Benjamin Weyori, Sampson, Daniel Ngala, Salifu Abdul - Mumin (Megabyte), Stephen Akobre, Juanita Pokuaa, Vida Ademin, Joshua Akanbasiam, too numerous to mention because of time and space constraints, that have helped me in one way or the other. Joshua Akanbasiam gets an extra acknowledgments for your brotherly love and care. For your love and care are unparallel.

Edward Baagyere KNUST, 2011

Contents

List of Tables

List of Figures

List of Acronyms

SWA	Smith - Waterman Algorithm
DNA	Deoxyribonucleic Acid
RSFC	RNS - SWA Forward Converter
WNS	Weighted Number System
ASIC	Application-Specific Integrated Circuit
GCD	Greatest Common Divisor
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
LSB	Least Significant Bit
MSB	Most Significant Bit
RNS	Residue Number System
CRT	Chinese Remainder Theorem
MRC	Mixed Radix Conversion
RRNS	Redundant Residue Number System

DCT	Discrete Cosine Transform
FCT	Fast Cosine Transform
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
ROM	Read Only Memory
MRD	Mixed Radix Digit
GCF	Greatest Common Factor
MR	Mixed Radix
MCRT	Modified Chinese Remainder Theorem
MATR	Matrix Method
CMOS	Complementary Meta Oxide Semiconductor
VHDL	VHSIC Hardware Description Language
CSA	Carry Save Adder
CPA	Carry Propagate Adder
EAC	End Around Carry
IMRC	Improved Mixed Radix Conversion
PE	Processing Element
LCM	Lowest Common Multiple
CLA	Carry Lookahead Adder
RSD	Redundant Signed Digit
BLAST	Basic Local Alignment Search Tool
FASTA	Fast Alignment Search Tool - All
MGAP	Micro Grained Array Processor
FSM	Finite State Machine
PLD	Programmable Logic Device
CPLD	Complex Programmable Logic Device
HDL	Hardware Description Language
BRC	Binary to RNS Converter

MUX	Multiplexer
GPU	Graphics Processing Units
CI	Custom Instruction
KCUPS	Kilo Cell Updates Per Second
LMB	Left - most bit
LUT	Look Up Table

Application of Residue Number System to Smith - Waterman Algorithm

Edward Yellakuor Baagyere

Abstract

In this thesis, we propose a hardware implementation of the Smith - Waterman Algorithm using Residue Number System (RNS).

One of the biggest challenges confronting the bioinformatics community as at now is fast and accurate sequence alignment. The Smith - Waterman algorithm (SWA) is one of the several algorithms used in addressing some of these challenges. Though very sensitive in doing sequence alignment, the SWA is not used in real life applications due to the computational cost involve in using this algorithm. Hence heuristics methods such as BLAST and FASTA are used though they do not guarantee accurate sequence alignments. We seek to address the computational challenge associated with the SWA by using the inherent arithmetic advantages of RNS. The RNS is such an integer system exhibiting the capabilities that support parallel computation, carry free addition, borrow-free subtraction, and single step multiplication without partial product. Base on some of these properties, a hardware implementation of the SWA is done in VHDL, a hardware description Language. In order to be able to use moduli set with a small dynamic range, matrix partitioning has been used on the fact that the comparison of two long strings of DNA can be done in a divide - and - conquer

approach. A sample 4 - bit system implementation improves the overall speed of the SWA. The implementation was on FPGA device EP2S15F484C3, a Stratix II family, and it consumes 189 logic cells. The worst - case clock - to - output delay (tco) between the specified source and destination points is 6.006 ns. These outstanding results when compared with what is in literature shows that the implementation is both area and speed efficient and thereby improve the speed constraints of the SWA. When compared with the work of L. Hasan and Z. Al-lrs (2007) this implementation has improves the computational time 243%, when calcualted in terms of percentage ratio, thereby making this implementation better than the state of the art hardware implementation of the SWA. In terms of comparison with the hardware implementation done by [1], our hardware implementation is superior by 143%.

Also, comparing our hardware implementation result with it's software equivalent shows a tremendous improvement of 871029%, that is 8710.29 times faster.

What these outstanding results mean is that there is hope for the bioinformatics community so far as accurate sequence alignment is concern. The total time that will be needed to alignment two strings of DNA using the RNS - SWA architecture will be improved by 8710.29 times more than it's software equivalent implementation.

These results also support the fact that RNS is a good platform to implement the SWA, since it has a high prospect of improving the overall computational cost and the hardware foot print of the algorithm.

Chapter 1 Introduction

1.1 Background

Bioinformatics is a field in life science that is gaining much attention in recent times and advances are made daily in this area of research. The social consequences of progress in this area are very enormous as the promise of finding cure to hitherto incurable diseases, prolonging life, and understanding the beginning and end of life are becoming more and more probable [2]. These prospects are achievable by the technique of sequence alignments.

The biological sequence alignments for sequence of *Deoxyribonucleic Acid* (DNA) or protein present an insight into the natural mutations occurring in the strings [3]. Also, similarities between two sequences might suggest evolution from the same genetic tree or mutations over time that occurred in one of the sequences in the given pair of sequences. Biological information from these sources are of significant importance to researchers in the field of bioinformatics, and therefore research in that direction cannot be over emphasized.

All organisms have cells and these cells consist of genetic information that make a particular organism different and unique from another organism. These genetic informations are carried by a chemical known as DNA in the nucleus of the cell. The DNA

of an organism consists of an interwoven strands that forms a *"double helix"*. Each strand is built from residues of molecules called *nucleotide*.

A nucleotide consists of two parts *viz: a phosphate group* and *a sugar group* called *deoxyribose*, these two parts form the ribbon-like backbone of the DNA strand and are identical in all nucleotides. There are four different kinds of bases, which define the four different nucleotides *viz:* Adenine (A), Cytosine (C), Guanine (G) and, Thymine

(T). The complete human genome contains approximately 3 million of these base pairs [4]. In order to discover the functional, structural and evolutionary relationship between two or more sequences of DNA, it is necessary to find the similarity between the sequences. This is done by finding the edit distance between the said sequence in question and the process is called *Sequence Alignment*.

There are several algorithms for doing sequence alignment. The commonly used ones are Fast Alignment Search Tool - All (FASTA) and Basic Local Alignment Search Tool (BLAST) [5]. FASTA and BLAST are fast algorithms which prune the search involved in a sequence alignment using heuristic methods, but they are not sensitive, that is, they don't guarantee exact alignments.

The Smith - Waterman Algorithm (SWA) [6] is very sensitive algorithm but it has a very high computational cost. Due to this high computational cost, the real life application of the SWA is much limited and the benefits that would have accrued from the field of bioinformatics are yet to come to the fore. For an example, the time and space complexity of this algorithm for comparing two sequences is $O(nm)$, where m and n are the lengths of the two sequences being compared. Although this computational complexity may not seem threatening, the growth in the genetic bio - sequence database is exponential. Thus the complexity that concerns the real world applications is really $O(knm)$, where k represents the exponential growth of the size in genetic databases [1].

The high computational cost of the SWA has a direct link with the carry propagation chains inherent to the Weighted Number Systems (WNS), e.g., binary number systems, decimal number systems. Because of this intrinsic performance limiter for arithmetic units and processors built based on WNS, several attempts have been made to over-

come the speed limitations by following two main research avenues as follows:

The carry propagation through the conventional ripple-carry adders, which is the main contributor to the addition delay, can be accelerated by using fast addition techniques [7]. Those make use of specialized circuitry able to de - serialize the carries calculation

via methods like carry look ahead (CLA), carry-skip, prefix calculation, anticipated calculation, etc. These fast addition techniques are very important in improving arithmetic units performance because other arithmetic operations such as multiplication and division are based on addition, thus their delay heavily depends on the addition delay. For non redundant number systems, e.g., traditional binary and decimal number systems, the delay of such fast adders is logarithmically bounded by the number of operand digits [8].

An alternative way to speed up the addition process is to make use of number system with specialized carry characteristics, i.e., proposing alternative number representation systems, e.g., Redundant Signed Digit (RSD) number representation systems [9–15] and Residue Number Systems (RNS) [14]. The sought characteristic of such alternative number systems is the capability to provide support for carry-free addition as this directly results in high-speed arithmetic units.

The RNS has many inherent interesting features. The RNS [8,16,17] is such an integer system exhibiting the capabilities to support parallel, carry-free addition, borrow-free subtraction, and single step multiplication without partial product. Moreover, it provides support for fault tolerance [18–20], which is becoming a crucial aspect as it is getting more and more expensive/difficult to fabricate perfect (predictable) devices in the context of deep sub-micron fabrication technology [21].

Due to some of these inherent features of RNS, it had wide spread application in Digital Signal Processing (DSP) applications, e.g., Digital Filtering [22–25], Convolutions, Correlations, Discrete Cosine Transform (DCT) [26, 27], Discrete Fourier Transform (DFT) [28–31], Fast Fourier Transform (FFT) [16, 24, 32, 33]. Additionally, RNS has also been applied in low power design [14,34–41], number theory [42–44], and digital communications [45–48].

With the increase in computing power and the rigorous research into computer number system, Residue Number System, and computer arithmetic, coupled with the decrease in the cost of memory, analyzing the enormous datasets generated by genome sequencing is becoming practicable in ways never thought possible. Even though such advances are encouraging and much needed, we are still several years away from the

immense amount of computing power that would be required to analyze these datasets completely, thus the need for further research in this field.

The RNS can be used to improve the performance of SWA. The SWA involves the basic RNS supported arithmetic operations such as addition, subtraction and multiplication. Since it has been shown in literature both theoretically and experimentally that using these basic arithmetic operations, RNS is faster than the conventional binary number system, we suggest RNS as an alternative candidate for improving the performance of the SWA.

1.2Problem Statement

The SWA is the most accurate sequence alignment algorithm available, but it is also the most expensive computationally, in particular for long sequences of DNA or protein [6]. Thus it guarantees exact matches between sequences, at the cost of long processing time. For example, when profiled on the MOLEN platform, a specific function within the SWA consumed 78% of the total run time [49]. Faster algorithms like FASTA and BLAST are available, but they achieve high speed at the cost of accuracy. The uncompromising computational cost of the SWA calls for a hardware acceleration of this algorithm using the inherent arithmetic advantages of Residue Number System (RNS) and that is what this research seeks to address.

1.3Objective of Study

The objectives of the research are the following under listed:

1.4. SPECIFIC OBJECTIVE

1. To solve the computational cost associated with the SWA by using computer arithmetics and RNS.

2. To further simplify the SWA and build a RNS-based SWA architecture with lower area footprint.

1.4 Specific Objective

To construct a RNS - based SWA processor using appropriate computer arithmetics techniques and RNS properties to solve the computational and area footprint associated with the SWA.

1.5 Justification

The SWA is a well-known algorithm for performing local sequence alignment; that is, for determining similar regions between two nucleotide or protein sequences. Instead of looking at the total sequence, the Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure.

One motivation for local alignment is the difficulty of obtaining correct alignments in regions of low similarity between distantly related biological sequences, because mutations have added too much “noise” over evolutionary time to allow for a meaningful comparison of those regions. Local alignment avoids such regions altogether and focuses on those with a positive score. Another motivation for using local alignments is that there is a reliable statistical model for optimal local alignments.

However, the SWA algorithm is fairly demanding of time and memory resources; in order to align two sequences of lengths m and n, $O(kmn)$ time and space are required. As a result, it has largely been replaced in practical use by the BLAST algorithm; although not guaranteed to find optimal alignments. These limitations therefore call for the hardware acceleration of this algorithm using the inherent arithmetic advantages of RNS, in order to explore the full potentials that the SWA has to offer to the bioinformatics community.

1.6Scope of Study

This research involves using RNS and computer arithmetics capabilities to solve the computational and space footprint that limits the real life application of the SWA.

1.7Design Methodology

The main objective of the research is to construct a RNS - based SWA processor to solve the computational and area footprint associate with the SWA.

This objective is achieved by using appropriate computer arithmetics and RNS properties. Further, the hardware implementation of the design is done using Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), a hardware programming language.

VHDL is a general - purpose hardware description language which can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. Also VHDL can describe a digital system at several different levels - behavioral, data flow and structural.

The results from the implementation are used to calculate the speed and area cost of the SWA and comparison is then made with the non - RNS based implementations in literature.

1.8 Organization of Thesis

This chapter provides an overview of the research undertaken in this thesis. The remainder of the work is organized as follows:

Chapter two, titled “Review of Literature”, summarizes works that have been done in the area of bioinformatics, sequence alignment algorithms, computer arithmetics, Residue Number System and, Data conversions types.

Chapter three, “Design and Simulation”, gives a detail description of the use of RNS and computer arithmetics to solve the computational and area cost associated with the SWA. The chapter further presents a hardware implementation of the RNS - based SWA architecture using VHDL and using *Quartus II version 4.0 VHDL* to compile and simulate the codes.

The next chapter is chapter four. It titled “Discussion of Simulation Results”, which deals with discussion of simulation results and then comparison between the RNS based SWA implementation and the non - RNS based is made. The thesis is completed with “Conclusion and Recommendation” which is captured under chapter five. This chapter gives a brief summary of the thesis findings and further research directions.

Chapter 2

Review of Literature

In this section, we seek to review literature on bioinformatics with key reference to sequence alignment methods, Smith - Waterman Algorithm (SWA), and the available hardware acceleration methods used to speed up the SWA. We shall also review literature on computer arithmetics, Residue Number System (RNS), and Data conversions types.

2.1 Bioinformatics

Bioinformatics is the application of information technology and computer science to the field of molecular biology. The term bioinformatics was coined by Paulien Hogeweg in 1979 [50] for the study of informatics processes in biotic systems. Its primary use since at least the late 1980s has been in genomics and genetics, particularly in those areas of genomics involving large-scale DNA sequencing.

Now, Bioinformatics entails the creation and advancement of databases, algorithms, computational and statistical techniques, and theory to solve formal and practical problems arising from the management and analysis of biological data. Over the past few decades rapid developments in genomics and other molecular research technologies and developments in information technologies have combined to produce a tremendous amount of information related to molecular biology. It is the name given to these

mathematical and computing approaches used to glean understanding of biological processes. Common activities in bioinformatics include mapping and analyzing DNA

and protein sequences, aligning different DNA and protein sequences to compare them and creating and viewing 3-D models of protein structures.

The primary goal of bioinformatics is to increase our understanding of biological processes. What sets it apart from other approaches, however, is its focus on developing and applying computationally intensive techniques (e.g., pattern recognition, data mining, machine learning algorithms, and visualization) to achieve this goal. Major research efforts in the field include sequence alignment, gene finding, genome assembly, protein structure alignment, protein structure prediction, prediction of gene expression and protein-protein interactions, genome-wide association studies and the modeling of evolution.

Bioinformatics was applied in the creation and maintenance of a database to store biological information at the beginning of the "genomic revolution", such as nucleotide and amino acid sequences. Development of this type of database involved not only design issues but the development of complex interfaces whereby researchers could both access existing data as well as submit new or revised data.

In order to study how normal cellular activities are altered in different disease states, the biological data must be combined to form a comprehensive picture of these activities. Therefore, the field of bioinformatics has evolved such that the most pressing task now involves the analysis and interpretation of various types of data, including nucleotide and amino acid sequences, protein domains, and protein structures. The actual process of analyzing and interpreting data is referred to as computational biology.

Some major research areas in Bioinformatics include:

- Sequence analysis
- Analysis of gene expression
- Analysis of regulation
- Analysis of protein expression

- Analysis of mutations in cancer
- Prediction of protein structure
- Comparative genomics
- Modeling biological systems
- Protein-protein docking

2.1.1 The Smith - Waterman Algorithm.

In 1981, T. F. Smith and M. S. Waterman described a method, commonly known as the Smith-Waterman(S-W) algorithm [6], for finding common regions of local similarity. The algorithm is a modification of the N - W algorithm which is a type of global sequence alignment. The algorithm is explained below:

In calculating the local alignment, matrix $H(i, j)$ is used to keep track of the degree of similarity between the two sequences to be aligned $A_i, B_j)$. Each element of the matrix $H(i, j)$ is calculated according to the following equation:

$H(i,0) = 0$, for $0 \geq i \leq m$

$H(0, j) = 0$, 0 for $0 \geq j \leq n$

$$H(i,j)=max\begin{cases}0 & \\ H(i-1,j-1)+S(i,j) & \text{match/mismatch in the diagonal}\\ H(i-1,j)-d & \text{deletion in sequence 1}\\ H(i,j-1)-d & \text{insertion in sequence 2}\end{cases} \quad (2.1)$$

$1 \geq i \leq m$,$1 \geq j \leq n$;

where:

$H(i, j)$ is the maximum similarity score between the two sequences.

$S(i, j)$ is the similarity score of comparing sequence A_i to sequence B_j and d is the gap penalty of mismatch

The algorithm consists of main three steps *viz:*

Table 2.1: The DP matrix and the trace back path.

		C	A	G	C	G	T	T	G
	0	0	0	0	0	0	0	0	0
A	0	0	2	0	0	0	0	0	0
G	0	0	0	4	2	2	0	0	2
G	0	0	0	2	3	4	2	0	2
T	0	0	0	0	1	2	6	4	2
A	0	0	2	0	0	0	4	5	3
C	0	2	0	1	2	0	2	3	4

1. Initialization step

2. Matrix fill step

3. Trace back step

The matrix is first initialized with $H(0, j) = 0$ and $H(i,0) = 0$,for all i and j· This is referred to as the *initialization step*. After the initialization, a matrix fill step is carried out using Equation 3.1, which fills out all entries in the matrix.

The third step is the trace back step, where the scores in the matrix are traced back to inspect for optimal local alignment. The third step, which is the trace, starts at the cell with the highest score in the matrix and continues up to the cell, where the score falls down to a predefined minimum threshold. In order to execute the trace back, the algorithm requires to find the cell with the maximum value, which is done by going through the entire matrix.

Table 2.1 shows the similarity between two sequences and the trace back in deep black. The time complexity of the initialization step is $O(M + N)$, where N and M are sizes of the two sequences. During the matrix fill step, the entire $H(i, j)$ matrix needs to be filled according to Equation 3.1, making its time complexity equal to the number of cells in the matrix or $O(MN)$. The time complexity of the traceback is also $O(MN)$, as the entire matrix needs to be traversed during this step. Thus the total time complexity of the SWA is $O(M + N)+O(MN)+O(MN) = O(MN)$. The total foot print of the

SWA is also $O(MN)$, as it fills a single matrix of size MN.

In order to reduce the $O(MN)$ complexity of the matrix fill stage, multiple entries of the $H(i, j)$ matrix are calculated in parallel. This is however complicated by data dependencies, whereby each (Hi, j) entry depends on the values of three neighboring entries $H(i, j-1)$,$H(i-1, j)$ and $H(i-1, j-1)$, with each of those entries in turn depending on the values of three neighboring entries, which effectively means that this dependency extends to every other entry in the region $H(\alpha,\gamma) : \alpha \leq i, \gamma \leq j$. This implies that it is possible to simultaneously compute all the elements in each anti diagonal, since they fall outside each others data dependency regions.

Apart from SWA, there are other local search methods such as FASTA (Fast Alignment Search Tool - All) and BLAST (Basic Local Alignment Search Tool). Based on heuristics, they are faster, although much less sensitive than the SWA.

2.1.2 Various Attempts to Accelerate the SWA in Hardware

Various approaches have been adopted to accelerate the SWA by implementing either the whole algorithm or some part of it in hardware and compare the performance with the software-only implementation.

Borah M. et al in 1994 [51] described the implementation of the SWA on a general purpose fine-grained architecture, the *Micro Grained Array Processor (MGAP)*. The authors show that their implementation is about 5 times faster than the rapid implementation of a genetic sequence comparator using filed programmable logic arrays. Their work was compared with that done by Daniel P. Lopresti, 1991 [52]. The work of Borah M. et al shows that parallel processor arrays, like MGAP, have the potential to solve computationally intensive problems in bioinformatics efficiently and less expensively.

In [53], the authors show the implementation of a fully custom processing unit to realize the execution of the SWA. The authors claimed that for conducting comparisons of multiple sequence pairs, using the same set of processing units, two approaches can be taken e.g. synchronous and asynchronous. The authors show that

the asynchronous parallel approach is (k-1)*(m-1) time steps faster than the synchronous parallel approach, where k represents the size of the existing sequences in the database, which grows exponentially.

Schroder A. et al in 2006, [54] demonstrated that the streaming architecture of the *Graphics processing Units (GPUs)* can be used for biological sequence database scanning. GPUs are single-chip processors, used primarily for computing 3D functions, but is also a good candidate for a bioinformatics applications such as sequence alignments. To achieve an efficient mapping on this type of architecture, the authors have formulated the SWA in terms of computer graphics primitives and claimed that the evaluation of their implementation on a high-end graphics card shows a speedup of almost sixteen compared to Pentium-IV, 3.0 GHz processor.

In 2005, Blas A. Di. et al [55] implemented the SWA on the Kestrel Parallel Processor for efficient query sizes. The Kestrel Parallel Processor is a single-board coprocessor with a 512-element linear array of 8-bit, SIMD processing elements. The performance was compared with the implementation on a 500 MHz, Ultra SPARC-II. The relative speed-up for a query size of 100 is reported to be seventeen. The other query sizes considered were 250 and 500. The speed -up achieved for the query size of 250was 49 times, whereas that for the query size of 500 was 99 times.

Laiq Hasan and Zaid Al-Ars in 2007, [1] divided the SWA into a number of functions, and then the time complexity of each function is measured, thus term known as code profiling. A software-only implementation of the SWA is profiled on Pentium-IV, 3.2 GHz processor, using the GNU profiler. The profiling results identify the most time consuming function. This function is then designed in VHDL. The processing run time of a software -only implementation on Pentium-IV, 3.2 GHz processor and hardware implementation on Virtex II Pro FPGA are compared to evaluate the percentage runtime improvement. The results show that the hardware implementation is 35.82 times faster than its equivalent software-only implementation.

In [56], Steve Margem use the power of reconfigurable computing to accelerate substantially the performance of the SWA. The percentage time spent on calculating the elements of the matrix, Hi,j, was cut down by nearly a third and the absolute time

spent on the algorithm dropped from 6,461 seconds to a little over 100 seconds, approximately 64 times faster than the equivalent software-only implementation on AMD Opteron processors.

Oliver T. et al in [57] showed a new approach to bio-sequence database scanning using reconfigurable FPGA-based hardware platforms to gain high performance at low cost. Their FPGA implementation achieves a speedup of approximately 170, as compared to a Pentium-IV, 1.6GHz processor.

Chiang J. et al in [58] also studied the improvement of the computational processing time of the SWA using *Custom Instructions (CIs)* on an FPGA board. This was done by first writing the SWA in pure software and replacing the portion which was the most computationally intensive with an FPGA custom instruction. Particularly, the designed CIs was on an Altera Nios II integrated development environment. The Nios II soft microprocessor was instantiate on an FPGA to allow rapid prototyping of new designs. Finally, they compared the processing runtime between the "pure software"and the "hardware acceleration"versions to calculate the percentage of runtime improvement. The results showed that the hardware accelerated algorithm improvement the processing runtime by an average of 287%. Thus using FPGA CIs is a promising direction for further research in improving genomic sequence searching.

In 2002, Yamaguchi Y. et al [59] proposed a high speed sequence alignment using run-time reconfigurable computing. With this approach, it is demonstrated that high performance can be achieved using off-the-shelf FPGA boards. The performance is almost comparable with dedicated hardware systems. The time for comparing a query sequence of 2048 elements with a database sequence of 64 millions elements by the SWA is about 34 seconds, which is about 330 times faster than a desktop computer with a Pentium-III, 1.0 GHz processor.

Yang B.H.W in 2002 [60], presented the design of a small custom processing element, called *Proclet*. This Proclet is used for a new VLSI implementation of the SWA. The results show that the design achieves a performance of 976 Kilo Cell Updates Per

Second(KCUPS), but is not compare with any reference design.

In the next section, we discuss the application of RNS to sequence alignment, an essential aspect of bioinformatics

2.2 Computer Number System and Arithmetics.

As the arithmetic applications grow rapidly, it is important for computer engineers to be well informed of the essentials of computer number systems and arithmetic processes.

With the remarkable progress in the very large scale integration (VLSI) circuit technology, many hitherto complex circuits that were unthinkable yesteryears become components easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetics, but also the unconventional ones are worth investigation in new design.

Numbers play an important rule in computer systems. Numbers are the basis and object of computer operations. Remarkably, the main task of computers is computing, which deals with numbers all the time.

Humans have been familiar with numbers for thousands of years, whereas the representation of these numbers in computer systems is a new issue. A computer can provide only finite digits for a number representation (fixed word length), though a real number may be composed infinite digits.

Because of the trade-offs between word length and hardware size, and between propagation delay and accuracy, various types of number representation have been proposed and adopted. In this section, we introduce the Conventional Radix Number System and Signed-Digit Number System, both belonging to Fixed-Point Number System, as well as Floating-Point Number System [14]. Additionally, the Residue

Number System (RNS) will be described with emphasis on its arithmetic advantages in real life application in bioinformatics.

2.2.1Conventional Radix Number System.

A *conventional radix number* N can be represented by a string of n digits such as $(d_{n-1}d_{n-2}...d_1d_0)_r$ with r being the radix. $d_i \leq i \leq n-1$, is a digit and $d_i \in \{0,1,...,r-1\}$, where the position of d_i matters, because 23 is a different number from 32. Such a number system is referred to as *positional weighted* system [9]. Mathematically,

$$N = d_{n-1}\cdot w_{n-1}+d_{n-2}\cdot w_{n-2}+...+d_0\cdot w_0 = \sum_{i=0}^{n-1} d_i\cdot w_i \qquad (2.2)$$

with d_i being the weight of position i. If r if fixed, as in the *fixed - radix number system* in our further discussion, $w_i = r^i$. Hence

$$N = d_{n-1}\cdot r^{n-1}+d_{n-2}\cdot r^{n-2}+...+d_0\cdot r^0 = \sum_{i=0}^{n-1} d_i\cdot r^i \qquad (2.3)$$

If r is not fixed, the number becomes a *mixed-radix number*.

To include the fraction into a fixed number N, let "." be a radix point with the integer part on left of it and fraction part on the right of it. There are n digits in the integer and k digits in fraction, such as $(d_{n-1}d_{n-2}...d_1d_0\cdot d_{-1}...d_{-k})_r$.

Then

$$N = \sum_{i=-k}^{n-1} d_i\cdot r_i \qquad (2.4)$$

In the string of weighted digits $(d_{n-1}...d_0\cdot d_{-1}...d_{-k})_r$, d_{n-1} is called the most significant digit (MSD), and d_k the least significant digit (LSD). A binary digit is referred to as a bit, and the above two digits are MSD and LSD, respectively. In an electric circuit, there are two voltage levels, "high"and "low", which can easily represent two digits, "1"and "0 ", in binary number system. More bits are required to represent a number in binary than in other radix systems. The number of bits required to encode a number ρ is blog_2ρc+1.

Here the *downstile* or *floor* of x bxc, is the greatest integer that is not greater than x, where x can be an integer or real. (Likewise, the *upstile* or *ceiling* of x dxe, is the smallest integer that is not smaller than x).

2.2.2 Conversion of Radix Numbers.

Computer systems recognize the binary, octal and hexadecimal numbers, however, humans who are mostly the end users of these numbers, are most familiar with the decimal number systems. Numbers can be converted from one radix system to another before, after or in the middle of arithmetic operations. We present below the algorithms for such conversions.

Given an integer,

$$(d_{n-1}d_{n-2}\cdots d_1d_0)_r,$$

with base r other than 10, such as $r = 2$ in binary, $r = 8$ in octal or $r = 16$ in hexadecimal, according to Equation 2.3, the following equations provides a method to convert it to the corresponding decimal number N_1·

$$N_1 = d_{n-1}\cdot r^{n-1}+d_{n-2}\cdot r^{n-2}+\cdots+d_0\cdot r^0 \tag{2.5}$$

That is, N_1 can be obtained by performing the multiplication of each given digit, the weight it carries and summing all the products.

In the reversed way, given a decimal number we can obtain the corresponding digits in its binary, octal or hexidecimal representation by division,using r as the divisor equal to 2,8 or 16, respectively.

Dividing both sides of Equation 2.5 by r, we obtained at the right - hand side the remainder d_0 and the quotient

$$d_{n-1}\cdot r^{n-2}+d_{n-2}\cdot r^{n-3}+\cdots+d_1, \tag{2.6}$$

since $d_0 < r$ and other terms on the right - hand side are integer times of r. if we divide the above quotient again by r, we will obtain the d_1, and so forth. After performing the $n-1$ times, d_{n-1} will become the quotient. If we divide it by r again, we will have quotient 0, since any $d_i < r$ and the last remainder d_{n-1}. The conversion procedure stops there.

Thus, to convert a decimal integer to a radix r number, the decimal number is initialized as the quotient. This quotient is repeatedly divided by r and the remainder is recorded until the quotient is zero. It should be noted that the LSB is generated first and the MSB is generated last.

On the other hand, for a radix r fraction number,

$$(0 \cdot d_{-1} d_{-2} \cdots d_{-k})_r,$$

with r 6= 10, the corresponding decimal number N_2 can be obtained by

$$N_2 = d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + \cdots + d_{-k} \cdot r^{-k}. \tag{2.7}$$

Also, a decimal fraction can be converted to a radix r number such as a binary, octal or hexadecimal number with r being 2, 8, 16, respectively.

Multiplying both sides Equation 2.7 by r, we have on the right - hand side

$$d_{-1} + d_{-2} \cdot r^{-1} + \cdots + d_{-k} \cdot r^{-k+1},$$

where d_{-1} is the integer part and others add up to the fraction part. Multiply the fraction part by r again, we have

$$d_{-2} + d_{-3} \cdot r^{-1} + \cdots + d_{-k} \cdot r^{-k+2},$$

where d_{-2} is the part. Continuing the multiplication process and retaining the digits in the integer part, the radix r number corresponding to a particular decimal fraction can be obtained. The conversion process is stopped when either the fraction part becomes

zero or a predefined precision is reached. The digits following the radix point from left to right with the integer digits, the earliest obtained first.

2.2.3Representation of Signed Numbers

As the signs of numbers are important and necessary for scientific computing, the representation of signed numbers is discussed below.

In the decimal number system the sign of a number is indicated by a + or − symbol to the left of the most - significant digit (MSD). In the binary number system, the *sign* of a number is denoted by the left - most bit (LMB), which is equal to 0 for a positive number, and 1 for a negative number.

In a general representation, let a conventional radix number A be an n digit signed number with the MSD representing the sign. That is,

$$A = (a_{n-1}a_{n-2}\cdots a_1a_0)_r,$$

and the *sign digit* a_{n-1} is decided as follows:

$$a_{n-1} = \begin{cases} 0 & \text{if } A \geq 0 \\ r-1 & \text{if } A < 0 \end{cases} \tag{2.8}$$

For an integer number, the radix point is on the right of a_0, that is,

$$(a_{n-1}a_{n-2}\cdots a_1a_0\cdot),$$

and that of a fraction number, the radix point is on the left of a_{a-2} such as

$$(a_{n-2}\cdot a_{n-2}\cdots a_1a_0)\cdot$$

In particular, when a_{n-1} 6= 0, we say that A is a *normalized fraction.*

In this review, we shall assume that A is an integer for our illustration. Let the magnitude of A be;

$$|A| = (m_{n-1} \cdots m_1 m_0)\cdot$$

If $a_{n-1} = 0$, A is a positive number, then,

$$A = (0a_{n-2} \cdots a_1 a_0)_r$$

$$= (0m_{n-2} \cdots m_1 m_0)_r\cdot$$

That is, number A has the same value as its true magnitude. Thus

$$A = \sum_{i=0}^{n-2} a_i r^i = \sum_{i=0}^{n-2} m_i r^i$$

If $a_{n-1} = r - 1$, A is a negative number, then the representation of the number will depend on which format to use.

There are three representation of negative numbers:

1. sign - magnitude,
2. diminished radix complement, and,
3. radix complement.

Sign - Magnitude Representation

In the familiar decimal representation, the magnitude of both positive and negative numbers is expressed in the same way. The sign symbol distinguishes a number as being positive or negative. This scheme is called the *sign - and - magnitude* representation.

The same scheme can be used with binary number system in which case the sign bit is 0 or 1 for positive or negative numbers, respectively. In general, the sign - magnitude

representation for a number A is $(r-1)m_{n-2}m_{n-3}\cdots m_1m_0$, where r is the radix of the number. For example, if we use four - bit numbers, then for $r = 2$, $-5 = (1)101$, and for $r = 10, -234 = (9)234$. Because of its similarity to the decimal sign representation, the sign - and - magnitude representation is easy to understand. However this representation is not well suited for use in computer arithmetic. More suitable number representations are discussed below.

Diminished Radix Complement

The general representation of the diminished radix complement is shown below;

$(r-1)\overline{m}_{n-2}\overline{m}_{n-3}\cdots m_1\overline{m}_0$,

where

$$\overline{m}_i = (r-1)-m_i, 0 \leq i \leq n-2\cdot$$

The diminished radix complement representation is also known as (r-1)'s complement denoted as

$$\overline{A} = r^n - 1 - |A|,$$

where n is the total number of digits including the sign digit. For example, given $r = 2$,

$$\overline{A} = 2^n - 1 - |A|,$$

and we have the 1^0s complement representation as follows: $-1010 : (1)0101\cdot$

Given $r = 10$, $\overline{A} = 10^n - 1 - |A|$, we have the following 9^0s complement representation.

$-7602_{10} : (9)2397\cdot$

Radix Complement

A general representation of a radix complement is $((r-1)m_{n-2}\overline{m}_{n-3}\cdots m_1 m_0)+\overline{1}$, where $m = (r-1)-m_i$, $0 \le i \le n-2$ The radix complement representation is also called *r's complement*, denoted as $\overline{A} = r^n - |A|$, For example, given $r = 2$, $\overline{A} = 2^n - |A|$, and we have the 2^0*s* complement representation as follows: $-1010_2 : (1)0110\cdot$ Given $r = 10$, $\overline{A} = 10^n - |A|$, and we have the following 10^0*s* complement representation: $-7602_{10} : (9)2398\cdot$

In the next few lines we shall discuss the representation of a fractional number using the radix complement format.

If $B = (0\cdot d_{-1}d_{-2}\cdots d_{-k})_r$, B is a positive number. It has the same value as the true magnitude of B. Thus

$$B = |B| = \sum_{j=-k}^{-1} d_j \cdot r^i,$$

comparing with Equation 2.4

$$N = \sum_{i=-k}^{n-1} d_i \cdot r^i,$$

$n = 0$ here.

If $B = ((r-1)\cdot d_{-1}d_{-2}\cdots d_{-k})_r$, B is negative. Then B has the following representations:

1. Sign - magnitude

   $(r-1)\cdot d_{-1}d_{-2}\cdots p_{-k}$,

2. Diminished radix complement

   $(r-1)\cdot \overline{d}_{-1}\overline{d}_{-2}\cdots \overline{d}_{-k}$, where $d_j = \overline{(r-1)-d_j}$, $-k \le j \le -1$. The diminished radix complement representation of B can be found by $\overline{B} = r^1 - r^{-k} - |B|$.

3. Radix complement

$\overline{\overline{\overline{\overline{((r-1)\cdot d_{-1}d_{-2}\cdots d_{-k})}}}}+r^{-k}$, where $d_j = (r-1)-d_j$, $-k \leq j \leq -1$.

The radix complement representation of B can be found by $\overline{B} = r^1-|B|$.

2.2.4The Residue Number System.

History of The Residue Number System.

The origin of Residue Number System (RNS) can be traced to the puzzle given by *Sun Tzu* [61], a Chinese Mathematician and is illustrated as follows: How can we determine a number that has the remainders 2, 3, and 2 when divided by the numbers 7, 5, and 3, respectively? This puzzle, written in the form of a verse in the third century book, *Suan -ching* by the Chinese scholar *Sun Tsu*, is perhaps the first documented use of number representation using multiple residues. The answer to this puzzle, 23, is outlined in *Sun Tzu's* historic work. The puzzle essentially asked us to convert the residues $(2|3|2)_{RNS(7|5|3)}$ into its decimal equivalent. *Sun Tsu* formulated a method for manipulating these remainders (also known as *residues*), into integers. This method is regarded today as the Chinese Remainder Theorem (CRT). The CRT, as well as the theory of residue numbers, was set forth in the 19th century by Carl Friedrich Gauss in his celebrated *Disquisitiones* Arithmetical [61].

This over 1700 - year - old number system is making waves in computing recently. Digital systems implemented on residue arithmetic units may play an important role in ultra - speed, dedicated, real - time systems that support pure parallel processing of integer - value data due to its inherent features such as carry free addition, borrow free subtraction, single step multiplication without partial product, parallelisms, and fault tolerant. These interesting properties of RNS have lead to its widespread usage in Digital Signal Processing (DSP) applications such as digital filtering, convolution, correlation, Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT), image processing, cryptography, communications, and other highly intensive arithmetic applications.

In the following, we give brief explanations on the above stated interesting inherent properties of RNS:

- RNS supports carry-free arithmetic operations. In the WNS, when performing addition, carries propagate from the Least Significant Bit(LSB) to the Most Significant Bit (MSB). Carries are also borrowed from the MSB to the LSB when performing subtractions. However,in RNS, unlike the WNS, carry-free additions and borrow-free subtractions are performed. Also, when multiplying numbers in the WNS, partial products, which must be added in order to obtain the final result, is generated whereas single step multiplication without partial product is carried out in RNS.

- RNS supports fast, parallel arithmetic operations. In RNS, digit by digit computations can be performed since there is no ordering significance between the digits. Thus, RNS supports parallel computations. As stated earlier, in RNS a weighted number is first broken down into a set of residues (also known as remainders). Arithmetic operations such as addition, subtraction, and multiplication are then performed on each of the residues simultaneously or, in parallel independent of one another. This advantage becomes even more apparent when the number of routine operations increases.

- RNS supports error detection and correction. The inherent properties of RNS suggest that a Redundant RNS (RRNS) can be used for self checking, error detection, and correction in digital processors. Error detection and correction is usually achieved by adding one or more redundant residue digits. As discussed earlier, there is no interaction between the residue digits, so any error that occurs in a single arithmetic module has a local effect and errors can easily be detected or corrected. In fact, the faulty module can be disconnected and the remaining modules redistributed between non-redundant and redundant digits. Many research work has been carried out on fault tolerance in RNS.

However, RNS has not found wide spread usage in general purpose processors due to difficulties associated with magnitude comparison, sign representation, overflow detection, data conversion, moduli selection, division, and other complex arithmetic operations.

2.2.5Definition of Residue Number System.

RNS is defined in terms of a relatively - prime moduli set $(m_1,m_2,...,m_L)$ that is *GCD* $(m_i,m_j) = 1$ for $i \neq j$, where *GCD* means *greatest common divisor*. A binary number X can be represented by the residues $(x_1,x_2,...,x_L)$, where $x_i = X \bmod m_i, 0 \leq x_i < m_i$. Such a representation is unique for any integer $X \in [0, M-1]$, where $M = \prod_{i=0}^{n-1} m_i$ is the dynamic range of the system. For a signed number system, any integer in $(-M/2, M/2)$ has a RNS n - *tuple* representation where $x_i = X \bmod m_i$ if $X > 0$, and $(M - |X|)$ otherwise. The signed RNS system is often referred to as a symmetric system. Addition, subtraction, and multiplication in RNS are very efficient since digit by digit computations are allowed. Additionally, there is no ordering significance between the digits. However, division in RNS is rather complex since it is not a closed operation. For example, given that X, Y, and Z have RNS representations [16]:

$$X \longrightarrow (x_1,x_2,...,x_L) \tag{2.9}$$

$$Y \xrightarrow{RNS} (y_1,y_2,...,y_L) \tag{2.10}$$

$$Z \xrightarrow{RNS} (z_1,z_2,...,z_L) \text{ and supposing that } op \text{ denotes} \tag{2.11}$$

the operation +, -, or *, then

$Z = X$ op Y, means

$$Z_i = (X_i op Y_i) \bmod m_i \tag{2.12}$$

^{RNS} if Z

belongs to Z_M.

This means that no carry information need be communicated between residue digits. This explains why RNS is applicable in high performance computing and thus widely used in highly intensive DSP applications. In order to fully exploit these RNS parallelisms, arithmetic units that efficiently implement the modular statement must be found.

Moduli selection and data conversion are one of the greatest challenges for RNS hardware design since the moduli choice affects the representational efficiency and the complexity of the arithmetic algorithms. To that end, a set of efficient moduli must be chosen and the moduli must be made as small as possible since it is the magnitude of the largest modulus that dictates the speed of the RNS arithmetic operations. Figure 2.1 shows that the n output words (corresponding to the number of moduli) that are generated by the binary to RNS converter (the front-end) are processed by the n-parallel processors in the RNS signal processor block producing n output words, which are converted to a conventional binary number by the RNS to binary converter (the back-end). Generally speaking, any RNS architecture must be interfaced efficiently

with a binary/decimal number system and for that purpose data conversions are required. As shown in Figure 2.1, the input operands must be first converted to RNS (forward conversion) and after the arithmetic operations have been performed, the output must be presented in the same way as the input (reverse conversion).

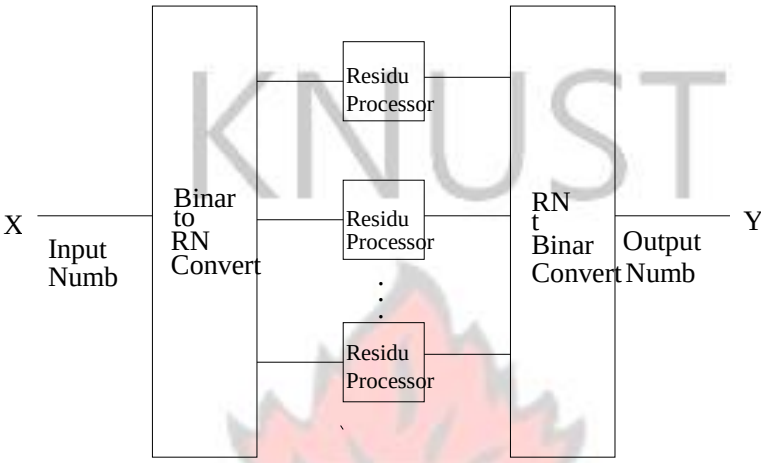


Figure 2.1: A typical RNS based Digital Signal Processor

In the next subsection, we briefly present Data Conversion and how the challenges associated with data conversion are addressed.

2.2.6Data Conversion

Data conversion is one of the greatest challenges of RNS because the input operands are provided in either standard binary or decimal format and must be converted to RNS before the computation can be performed. Similarly, the final results must be represented in the same way as the input operands, thus RNS to binary/decimal conversion is very essential to a successful RNS design.

This implies that RNS based processors make heavy use of data conversions, which are slow processes. For an RNS processor to compete favorable with a conventional processor efficient data converters must be developed so that the RNS speedup will

not be nullified by the conversion overhead. Data conversion can be divided into two categories, namely- forward and reverse conversion. Relatively, the reverse conversion is more complex but the forward conversion is not simple either. In the next section, we provide simple explanations on each of these two categories.

2.2.7 Forward Conversion

The input operands to the RNS processor are either in the decimal or binary format, and therefore need to be converted into their respective residues before they are used for the computation. This work of converting from decimal/binary to residue is done by the forward converter.

For any n - bit nonnegative integer X in the range $0 \leq X \leq 2^n - 1$ can be represented in the weighted binary system as

$$X = \sum_{i=0}^{N-1} b_i 2^i \tag{2.13}$$

where $b \in (0,1)$

The binary value of X can be converted into a set of n residues as x , where $x_i = X \bmod m_i$. The values of x_i can be found by the following steps:

From equation 4,

$$X = \sum_{i=0}^{N-1} b_i 2^i \tag{2.14}$$

$$\text{Let } X \bmod m_i = |X|_{m_i} \tag{2.15}$$

This implies

$$|X|_{m_i} = \left| \sum_{i=0}^{N-1} b_i \left| 2^i \right|_{m_i} \right|_{m_i} \tag{2.16}$$

The term $|2^i|_{m_i}$ can be pre - computed and stored in a Look Up Table (LUT). Also for any n - bit signed integer X in the range $0 \leq X \leq 2^n - 1$, the residues of X can be represented in the 2^0s - complement form as;

$$= \left|$$

$$X_{2\text{0scompl}} = (b_n b_{n-1} \ldots b_1 b_0) = -b_n 2_n + \sum_{1=0}^{n-1} b_i 2_i \left| \right| \quad (2.17)$$

Let $x_i = X \bmod m_i$ (2.18)

Then $x_i \quad b_n m_i - |2^n|_{mi}) + \sum b_i |2^i|_{mi} \left| \right|_{1=0m_i}$ (2.19)

Again the value of $|2^i|_{mi}$ can be pre - computed and stored in a LUT.

2.2.8The Reverse Conversion

Several reverse conversion techniques have been proposed in literature based on either the traditional Chinese Remainder Theorem (CRT) or the Mixed Radix Conversion (MRC) which may or may not rely on LUTs. The CRT is desirable because the data conversion can be parallelized while MRC is a sequential process by its very nature. However, many up to date RNS to binary/decimal converters are based on MRC due to the complex and slow modulo - M operation (where M is the system dynamic range, thus a rather large constant) required by CRT. In the next subsection, we present some necessary information about CRT.

2.2.9The Chinese Remainder Theorem

The magnitude of RNS number can be obtained from the CRT formula:

$$X = (x_{L-1}|\ldots|x_2|x_1|x_0) = \left|\sum_{i=0}^{L} s_i \left|\left(x_i s_i^{-1}\right)\right|_{m_i}\right|_M \quad (2.20)$$

where $s_i = \frac{M}{m_i}$ and s_i^{-1} is the multiplicative inverse of $|s_i|_{m_i}$

which implies that $\left|s_i s_i^{-1}\right| = 1$

Figure 2.2 gives the schematic diagram of the CRT. This diagram clearly shows the inherent parallelism feature of the CRT [16]. The traditional CRT can be further simplified when certain moduli sets (whether relatively prime or not) are utilized [62–64]. Recently, CRT that requires $mod\ -s_2s_3...s_L$ instead of $mod\ -s_1s_2...s_L$ required by the traditional CRT has been reported. This is called the New CRT and is presented in [65]. Based on the New CRT, many efficient reverse converters have been presented [62–64].

We briefly review the New CRT as follows [66]:

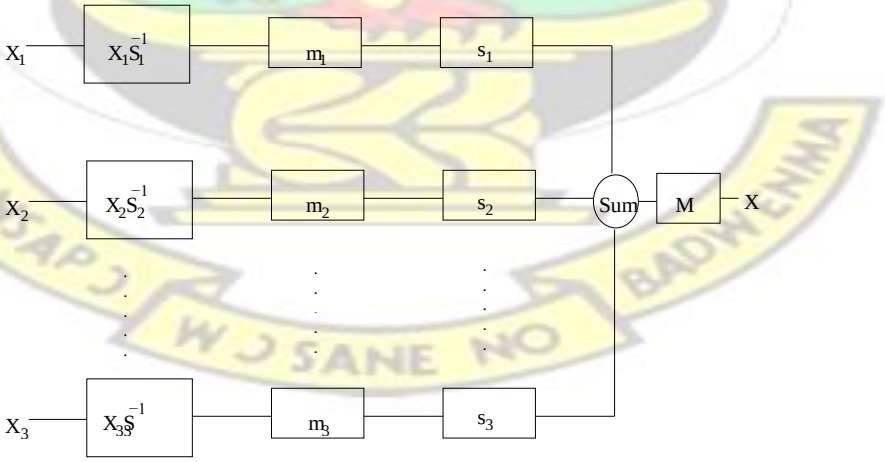


Figure 2.2: A schematic diagram of the CRT

Given the residue number$(x_1,x_2,...,x_L)$ with respect to the moduli set $(s_1,s_2,...,s_L)$, the corresponding decimal number X is computed as:

$$X = x_1+|k_1s_1(x_2-x_1)k_2s_1s_2(x_3-x_2)+\cdots+k_{L-1}s_1s_2...s_{L-1}(x_L-x_{L-1})|_{s_2s_3...s_L}$$

The main drawback of CRT emerges from the required modulo-M operation which, given that M is a rather large number, can be time consuming and rather expensive in terms of area and energy consumption. The MRC is an alternative method which does not involve the large modulo-M calculations.

2.2.10 The Mixed Radix Conversion.

Conversion from RNS to decimal is relatively fast using Mixed Radix Conversion (MRC) as it does not involve the large modulo-M calculations present in CRT.

Suppose that we have an RNS number $(x_1,x_2,...,x_L)$ with the corresponding set of moduli$(m_1,m_2,...,m_L)$ and a set of digits$(a_1,a_2,...,a_L)$ which are the Mixed Radix Digits *(MRDs)*, the decimal equivalent of the residues can be computed as follows:

$$X = a_1+a_2m_1+a_3m_1m_2+\cdots+a_nm_1m_2m_3...m_{n-1} \quad (2.21)$$

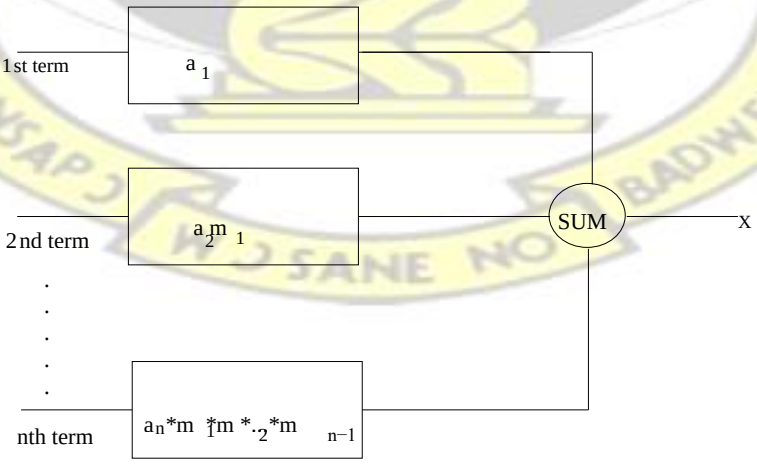


Figure 2.3: A schematic diagram of the MRC

where the the mixed radix digits are given as follows:

$$
\begin{aligned}
a_1 &= x_1 \\
a_2 &= \left|(x_2 - a_1)\left|m_1^{-1}\right|_{m_2}\right|_{m_2} \\
a_3 &= \left|\left((x_3 - a_1)\left|m_1^{-1}\right|_{m_3} - a_2\right)\left|m_2^{-1}\right|_{m_3}\right|_{m_3} \\
&\vdots \\
a_n &= \left|\left(\left((x_n - a_1)\left|m_1^{-1}\right|_{m_n} - a_2\right)\left|m_2^{-1}\right|_{m_n} - \cdots - a_{n-1}\right)\left|m_{n-1}^{-1}\right|_{m_n}\right|_{m_n}
\end{aligned}
\tag{2.22}
$$

Figures 2.4 and 2.3 show how the decimal equivalent and the MRDs of the residues can be computed respectively [63]. For the *MRD* a_i, $0 \leq a_1 < m_i$ any positive number in the interval $[0, \prod_{i=1}^{n} m_i - 1]$ can be uniquely represented. The only obstacle with the *MRC* is that by its very nature, it is a sequential process. Several attempts have been made to address this short-coming [67–69]. As stated earlier, data conversion and moduli selection are the two most important issues for a successful RNS design. Thus, this review will be incomplete without discussing moduli selection. Consequently, moduli selection is the subject of next section.

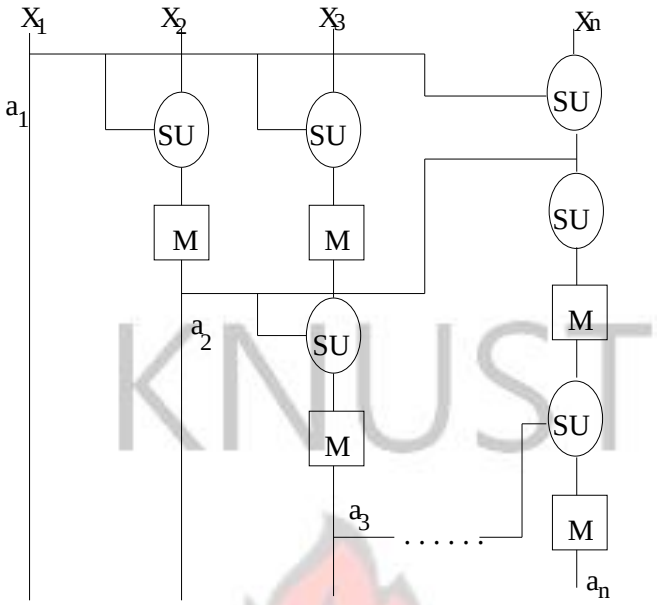


Figure 2.4: MRDs schematic diagram

2.2.11 Moduli Selection

The forms and the number of moduli selected determine the speed, the dynamic range, and the hardware complexity of the resulting RNS architecture. The magnitude of the largest modulus dictates the speed of the arithmetic operations. The moduli selected should therefore be made comparable in magnitude since there is no advantage in further fragmentation as the speed is already being dictated by the magnitude of the largest modulus. We present the classification of moduli selection in the following subsections.

2.2.12 Unrestricted Moduli selection

Unrestricted moduli selection, which is one of the categories of moduli selection, is presented first while the restricted moduli selection, the second category of moduli selection is the topic of next subsection.

In an unrestricted moduli selection, prime numbers are chosen in sequence until the desired dynamic range M is obtained. Thus mutually prime integer moduli are the topmost among the RNS researchers' design considerations. However, unrestricted moduli selection does not support simple conversions and simple RNS arithmetic

computations. The solutions for realizing all arithmetic operations are based on ROMs, in order to speed up execution. The cost of implementing ROM based RNS data converters is generally very high. Thus, the need for restricted moduli selection [14, 70–72], which is discussed next.

2.2.13Restricted moduli Selection

These restricted moduli sets are based on powers - of - two related moduli. This class of moduli set eliminates the need for ROMs in building RNS data converters [73]. Additionally, with the restricted moduli sets, the basic building blocks such as multipliers,adders, binary - to - RNS converters, and RNS - to - binary converters can also be easily realized using logic gates. Again, using restricted moduli sets, several adder based data converters have been proposed. For example,
$\{2^n-1,2^n,2^n+1\},\{2^n-1,2^n,2^{n-1}-1\},\{2n-1,2n,2n+1\},\{2n,2n+1,2n+2\}$ In general, moduli sets have some unique common features [70, 74]. These features are discussed next.

2.2.14Expected Features of the Moduli Sets.

In selecting the moduli set $\{m_i s\}_{i=1-L}$ the following general rules are considered:

1. They should be relatively prime;
2. The moduli $m_i s$ should be made as small as possible so that operations modulo require minimum computational time;
3. The moduli $m_i s$ should imply simple weighted to RNS and RNS to weighted conversions as well as simple RNS arithmetic. Sets with all their moduli being of the forms $2^{n1}+1$ and $2^{n2}-1$ and one of the form 2^{n3} satisfy the requirement of simple conversions and simple arithmetic.

4. The dynamic range should be large enough in order to avoid overflow;
5. The moduli $m_i s$ should create a balanced decomposition of the Dynamic Range. This implies that the differences between the number of bits of the different moduli should be very small;

In the previous sections, we have presented a number of issues which are of paramount importance to the design of RNS based processors. The next section talks about the application of these arithmetic advantages of the RNS to bioinformatics in order to speed up the computational challenge of the SWA.

2.2.15 Application of Residue Number System to Bioinformatics.

The arithmetic advantages associated with these number systems are employed to address the computational challenge of the SWA in the next chapter. As we know, the main tasks of a computer is computing which deals with numbers all the times. The type of the number system used greatly impacts the performance and area of a digital system, and must therefore be carefully looked at when implementing any thing that has to do with computing.

Chapter 3

Design and Simulation

In this chapter, we outline the digital system implementation process and how this process is used in the implementation of the Residue Number System - Smith - Waterman Algorithm (RNS-SWA) architecture. The inherent features of RNS as shown in Section 2.2.4 are used to speed up the computational challenge of the SWA.

3.1 Digital System Implementation Process

In general, a digital system is a sequential circuit made up of interconnected flip flops and gates. The system is partitioned into modular subsystems, each of which performs some functional task. The modules are constructed hierarchically from functional blocks such as registers, counters, decoders, multiplexers, buses, arithmetic elements, flip - flops, and primitive gates.

Interconnecting the various subsystems through data and control signals results in a digital system. In this digital system, we partition the system into two types of modules:

- datapath, and
- control unit.

3.1.1 Datapath

The Datapath is responsible for all the operations performs on the data. It includes:

- Functional units such as adders, shifters, multipliers, Arithmetic and Logic Units (ALUs) and, comparators.
- Registers and, other memory elements for temporary storage of data and,
- Buses and, multipliers for the transfer of data between the different components in the datapath through the data input lines. Results from the computation are returned through the data output lines.

3.1.2 Control Unit

The control unit (controller) is responsible for controlling all the operations of the data path by providing appropriate control signals to the datapath at the appropriate times. At any one time, the control unit is said to be in a certain state as determine by the content of the state memory. The state memory is simply a register with one or more (D) flip - flops.

The control unit operates by transitioning from one state to another - one state per clock cycle, and because of this behavior, the control unit is also referred to as finite - state machine (FSM). The next - state logic in the control unit will determine what state to go to next in the next clock cycle depending on the current state that the FSM is in, the control inputs, and the status signals.

In every state, the output logic that is in the control unit generates all the appropriate control signals for controlling the datapath. The datapath, in return, provides status signals for the next - state logic. Upon compilation of the computation, the control output line is asserted to notify external devices that the value on the data output lines is valid.

The Control signals are binary signals that activate the various data - processing operations. To activate sequence of such operations, the control unit sends the proper sequence of control signals to the datapath.The control unit, in turn, receives status bits from the datapath [75].

These variables describe aspects of the state of the datapath. The control unit uses the variables in defining the specific sequence of the operations to be performed.

In implementing a digital system, we made use of Programmable Logic Device (PLD), these include Simple Programmable Device, Complex Programmable Logic Device (CPLD), and Field Programmable Gated Array (FPGA). This can be thought of as a “blank slate” on which you implement a specified circuit or system design using a certain process. This process requires

a software development package installed on a computer to implement a circuit design in the programmable chip. The computer must be integrated with a development board or programmable fixture containing the device.

Any digital logic design based on PLD must pass through several design steps called the *design flow* as shown in Figure 3.1. The constituents of the module design flow are explained below.

3.1.3 The Module Design Flow

- Design Entry: This is the first programming step. The circuit or system design must be entered into the design application software using text - based, graphic entry (schematic capture), or state diagram description. Design entering is device independent. Text - based entry is accomplished with a hardware description language (HDL) such as VHDL, Verilog, AHDL, or ABEL. Graphic (Schematic) entry allows pre-stored logic functions from a library to be selected, placed on the screen, and then interconnected to create a logic design. State diagram entry requires specification of both the states through which a sequential logic circuit progresses and the conditions that produce each state change.

  Once the design has been entered, it is compiled. A compiler is a program the controls the design flow process and translates source code in format that can be logically tested or downloaded to a target device. The source code is created

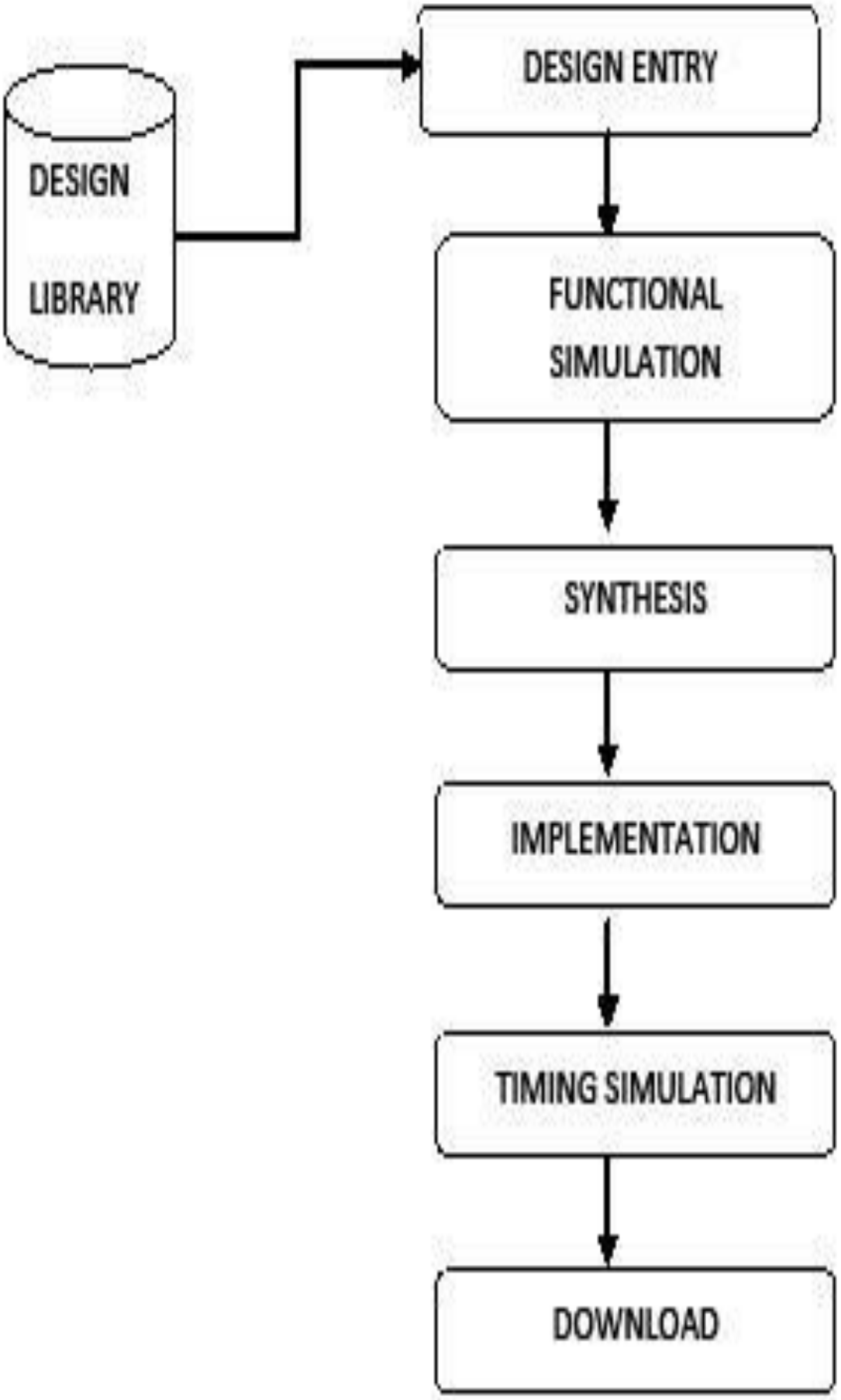

DESIGN ENTRY
DESIGN
LIBRARY
FUNCTIONAL
SIMULATION
SYNTHESIS
IMPLEMENTATION
TIMING SIMULATION
DOWNLOAD

Figure 3.1: The Module Design Flow Diagram

during design entry, and the object code is the final code that actually causes the design to be implemented in the programmable device.

- Functional simulation: The entered and compiled design is simulated by software to confirm that the logic circuit functions as expected. The simulation will verify that correct outputs are produced for a specified set of inputs. A device independent software tool for doing this is generally called a *waveform editor*. Any flaws demonstrated by the simulation would be corrected by going back to design entry and making appropriate changes.

- Synthesis: Synthesis is where the design is translated into a netlist, which has a standard form and is device independent.

- Implementation: Implementation is where the logic structures described by the netlist are mapped into the actual structure of the specific device being programmed. The implementation process is called fitting or place and route and results in an output called a bit - stream, which device dependent.

- Timing simulation: This step comes after the design is mapped into the specific device. The timing simulation is basically used to confirm that there are no design flaws or timing problems due to propagation delays.

- Download: Once a bitstream has been generated for a specific programmable device, it has to be downloaded to the device to implement the software design in hardware. Some programmable devices have to be installed in a special piece of equipment called a *device programmer* or on a development board.

3.2 Hardware Implementation of the RNS - SWA Architecture

In this section we give a brief description of the SWA and its pseudo code. The section also presents the hardware implementation procedures of the RNS-SWA architecture.

3.2.1 The Smith - Waterman Algorithm

The algorithm is explained below:

In calculating the local alignment, matrix $H(i, j)$ is used to keep track of the degree of similarity between the two sequences to be aligned, that is A_i and B_j. Each element of the matrix $H(i, j)$ is calculated according to the following equation:

$$H(i,j) = max \begin{cases} 0 \\ H(i-1, j-1) + S(i,j) & \text{Diagonal Entry} \\ H(i-1, j) - d & \text{Upper Entry} \\ H(i, j-1) - d & \text{Left Entry} \end{cases} \quad (3.1)$$

where:

$H(i, j)$ is the maximum similarity score between the two sequences.

$S(i, j)$ is the similarity score of comparing sequence A_i to sequence B_j and d is the gap penalty of mismatch

Diagonal, Upper and Left entries are the matrices entry position relative to the current $H(i, j)$ calculation.

The pseudo code of the SWA is also shown below:

As stated in Chapter 2 under Section 2.1.1, the SWA computation involves three main steps. The matrix fill step is computationally intensive and it is this very step that had been accelerated in hardware by various researchers.

In this section, we give a detail procedure of the hardware implementation of this step using RNS as tool, making use of its carry free, modularity and one step multiplication

```
1  Declare an nxm similarity matrix;
2  Initialize the top row (i = 0) and left column (j = 0) with 0;
3  for i = 1;i < length(Sequence); i++ do
4      for j = 1; j < length(Sequence); j++ do
5          H(i,j) = max{0,H(i−1, j−1)+S(i, j),H(i−1, j)−d,H(i, j−1)−d};
6      end
7  end
8  Save index of term that contributed to the calculated value in H(i,j);
9  Find maximum value in nxm matrix;
10 Using saved indices in 8, traceback to find 0 encountered;
```

Algorithm 1: The Pseudo code of the Smith - Waterman Algorithm

features as outline in Section 2.2.4 above. The complete implementation of the SWA involves basically three steps. These include:

1. The Binary/Decimal to RNS Conversion stage. This step is christened the RNS Based SWA Forward converter (RSFC).

2. The RNS based arithmetic operations stage. This is also termed the RNS base SWA microprocessor stage and,

3. The RNS magnitude comparison stage.

A block diagram of the RNS-SWA architecture is shown in Figure 3.2 with these three stages layout. In the next section these three stages are implemented on a PLD system employing the inherent arithmetic properties of RNS.

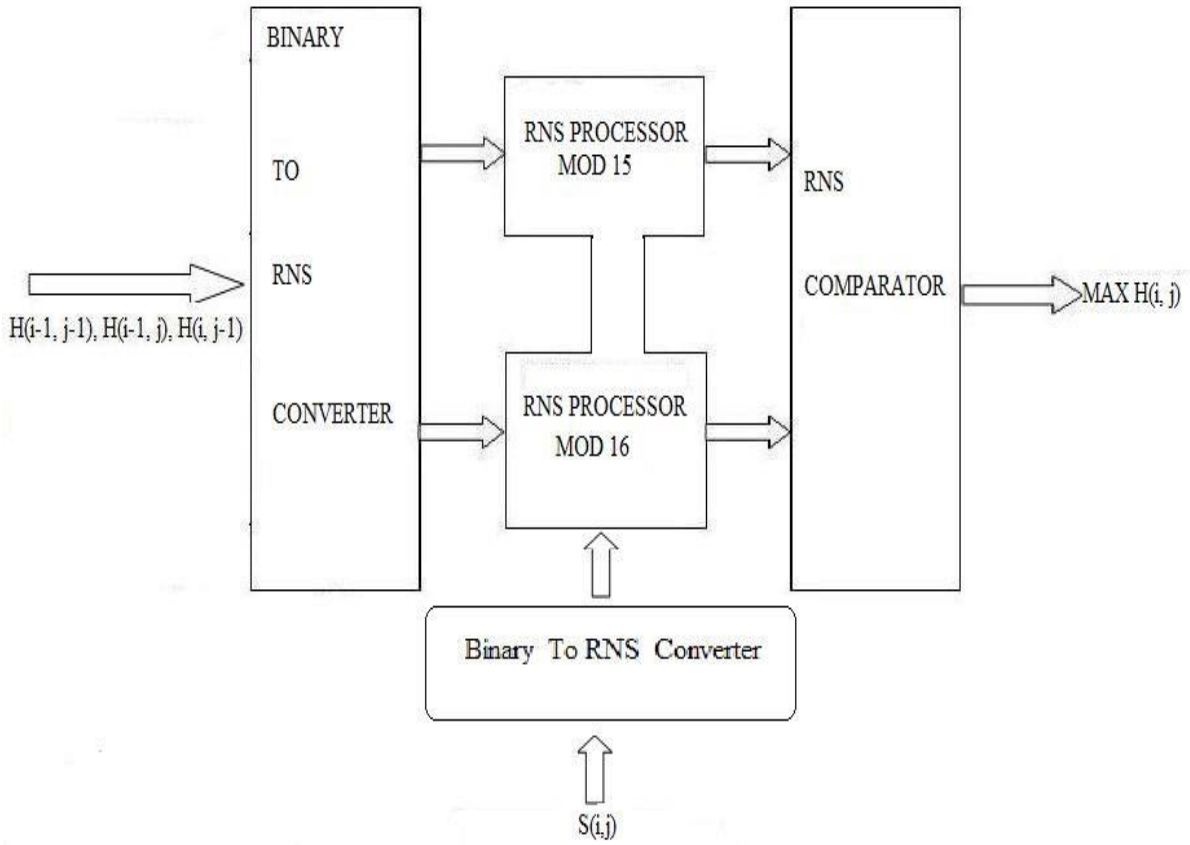


Figure 3.2: The RNS-SWA Architecture

3.2.2 The RNS-SWA Forward Converter

In this section, we shall describe the Residue Number System Based Smith-Waterman Algorithm Forward Converter (RSFC), which is one of the components of the hardware implantation of the RNS - SWA based architecture.

The RSFC is made up of the binary/decimal input values which comprise of $H(i-1, j-1)$, $S(i, j)$, $H(i-1, j)$, $H(i, j-1)$ and d, as shown in the equation 3.1. These binary/decimal values are converted into residues numbers by the Binary to RNS Converter (BRC), which is termed the RNS forward conversion. The residues produced are then used to execute carry free addition, borrow free subtraction by the two RNS processors as shown in Figure 3.2. Each of the residue processors does concurrent data processing, independent of each other, and thereby speeding up the arithmetic operation involves in the calculation of the SWA.

Based on the architecture shown in Figure 3.2, we present a customized memoryless RNS forward converter using combinational logic; it does not need any memory or Processing Elements (PEs) in its residue computation. The converter also works for both signed and unsigned numbers.

3.2.3Selection of Moduli Set

As outline in Section 2.2.11, the forms and the number of moduli selected determine the speed, the dynamic range, and the hardware complexity of the resulting RNS architecture. The magnitude of the largest modulus dictates the speed of the arithmetic operations.

In our work, we choose to use the moduli set $\{2^n, 2^n-1\}$ for the following reasons: firstly, it provides simpler designs for converters and magnitude - related operations, thus is more applicable to our design; secondly, it is the most commonly used moduli set in literature, using this moduli set makes our work comparable to most existing designs.

In order to be able to use a moduli set with this smaller dynamic range, matrix partitioning has been used based on the fact that the comparison of two long strings can be done in a divide-and-conquer fashion. The moduli set used in the implementation is $m = \{2^n, 2^n-1\}$, where $n = 4$ and, $m = \{16,15\}$, with a dynamic range of $M = 240$.

The elements of this moduli set within the given dynamic range for signed numbers are shown in Table 3.1

3.2.4The Memoryless RNS - SWA Forward Converter

As most existing devices and applications use binary representations, such as fixed point or floating - point numbers, the first part of an RNS design is usually a converter

that converts binary numbers into residues format, which is usually termed as the forward converter or residue generator.

Table 3.1: The Residues Table for Mod 16 and Mod 15 for signed numbers in hexadecimals

Decimal Number	Hexadecimal Number	(Mod 16, Mod 15)
-120	88	(8, 0)
-119	89	(9, 1)
-118	8A	(A, 2)
...	...	...
-3	FD	(D, C)
-2	FE	(E, D)
-1	FF	(F, E)
0	00	(0, 0)
1	01	(0, 0)
2	02	(2, 2)
3	03	(3, 3)
...	...	...
117	75	(5, C)
118	76	(6, D)
119	77	(7, E)

Early efforts [16,23,61,76] on forward converters decompose the binary value into an array of power - of - two values, and sum them up with modular adders, this concept is outline in Section 2.2.7

In our implementation of the Memoryless Forward Converter, with the use of this specific moduli set $m = \{2^n, 2^n - 1\}$, makes the generation of the residue values greatly simplified, just by the use of combinational logic without use of any memory. The implementation steps outline below were used in accordance with the design flow diagram shown in Figure 3.1.

1. The Design Entry: The memoryless RNS forward converter is entered into a Quartus II version 4.0 VHDL application software using the graphic entry or schematic capture tool embedded in the software. This process allows pre-stored logic functions from the software library to be selected, placed on the screen, and

then interconnected to create the logic design. The implemented diagram of the converter is shown in Figure 3.3.

A decimal number, D, which is an 8 - bit number is partitioned into two nibbles namely *U* and *V*. U (i.e. U0, U1, U2, U3) is the high order nibble of the binary

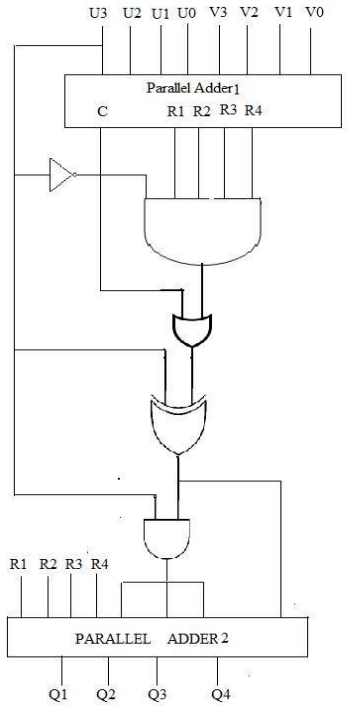


Figure 3.3: The Memoryless RNS - SWA Forward Converter

representation of D and V (i.e. V0, V1, V2, V3) is the low order nibble. For modulus 2^n, the residue is simply the lowest order n bits. However for modulus $2^n - 1$, the forward conversion is not as simple.

In the case of modulus 15 the high order nibble is added to the low order nibble by a parallel adder which is made up of one half adder and three full adders, labeled Parallel Adder1 in Figure 3.3. The sum from Parallel Adder1, namely R1, R2, R3, R4, forms an operand for a second stage of addition. The second operand for this stage is derived from U3, the carry-out and the sum from Parallel Adder1 as per the intervening logic shown between Parallel Adder1 and Parallel Adder2 in Figure 3.3. The sum from this stage (without the carry-out) constitutes the D mod 15 representation. D mod 16 is simply the nibble V.

2. Compilation: After the design entry is completed, it is compiled, in order to translate the source object code into object code in a format that can be logically tested or downloaded to a target device.

3. Functional Simulation: The next step that follows the compilation process is the functional simulation. This is done by the software to confirm that the logic circuit functions as expected. The simulation will verify that correct outputs are produced for a specified set of inputs, and it is the waveform editor (a device independent software tool) that is used to verify this.

4. Timing Simulation: Finally, timing simulation was done to verify that the circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the hardware device.

The output of Parallel-Adder2 and the four LSBs of the decimal number are the result of the RNS representation of the decimal number in question.

The VHDL codes of the memoryless forward converter are shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work; ENTITY
MODULUS15_16 IS port (
V0 : IN STD_LOGIC;
V1 : IN STD_LOGIC;
V2 : IN STD_LOGIC;
V3 : IN STD_LOGIC;
U0 : IN STD_LOGIC;
U1 : IN STD_LOGIC;
U2 : IN STD_LOGIC;
U3 : IN STD_LOGIC;
P0 : OUT STD_LOGIC;
```

```
P1 : OUT STD_LOGIC;
P2 : OUT STD_LOGIC;
P3 : OUT STD_LOGIC;
Q0 : OUT STD_LOGIC;
Q1 : OUT STD_LOGIC;
Q2 : OUT STD_LOGIC;
Q3 : OUT STD_LOGIC
);
END MODULUS15_16;
ARCHITECTURE bdf_type OF MODULUS15_16 IS component and_5
PORT(data0 : IN STD_LOGIC; data1 : IN STD_LOGIC; data2 : IN
  STD_LOGIC; data3 : IN STD_LOGIC; data4 : IN STD_LOGIC;
  result : OUT STD_LOGIC
); end component; component paralell_adder2b
PORT(R0 : IN STD_LOGIC;
  R1 : IN STD_LOGIC;
  R2 : IN STD_LOGIC;
  R3 : IN STD_LOGIC;
  F0 : IN STD_LOGIC;
  F1 : IN STD_LOGIC;
  F2 : IN STD_LOGIC;
  F3 : IN STD_LOGIC;
  A0 : OUT STD_LOGIC;
  A1 : OUT STD_LOGIC;
  A2 : OUT STD_LOGIC;
  A3 : OUT STD_LOGIC ); end component; component
parallel_adder1
PORT(V0 : IN STD_LOGIC;
  V1 : IN STD_LOGIC;
  V2 : IN STD_LOGIC;
  V3 : IN STD_LOGIC;
```

```
  U0 : IN STD_LOGIC;
  U1 : IN STD_LOGIC;
  U2 : IN STD_LOGIC;
  U3 : IN STD_LOGIC;
  R0 : OUT STD_LOGIC;
  R1 : OUT STD_LOGIC;
  R2 : OUT STD_LOGIC;
  R3 : OUT STD_LOGIC;
  C0 : OUT STD_LOGIC ); end component; signal SYNTHESIZED_WIRE_17 :
STD_LOGIC; signal SYNTHESIZED_WIRE_18 : STD_LOGIC; signal
SYNTHESIZED_WIRE_19 : STD_LOGIC; signal SYNTHESIZED_WIRE_20 :
STD_LOGIC; signal SYNTHESIZED_WIRE_4 : STD_LOGIC; signal
SYNTHESIZED_WIRE_5 : STD_LOGIC; signal SYNTHESIZED_WIRE_6 :
STD_LOGIC; signal SYNTHESIZED_WIRE_7 : STD_LOGIC; signal
SYNTHESIZED_WIRE_21 : STD_LOGIC; signal SYNTHESIZED_WIRE_22 :
STD_LOGIC; BEGIN

P0 <= V0;
P1 <= V1;
P2 <= V2; P3 <= V3;
b2v_inst : and_5
PORT MAP(data0 => SYNTHESIZED_WIRE_17,
  data1 => SYNTHESIZED_WIRE_18, data2 =>
  SYNTHESIZED_WIRE_19, data3 =>
  SYNTHESIZED_WIRE_20, data4 =>
  SYNTHESIZED_WIRE_4, result =>
  SYNTHESIZED_WIRE_6);
SYNTHESIZED_WIRE_4 <= NOT(U3);
SYNTHESIZED_WIRE_7 <= SYNTHESIZED_WIRE_5 OR
SYNTHESIZED_WIRE_6;
```

```
SYNTHESIZED_WIRE_21 <= SYNTHESIZED_WIRE_7 XOR U3;
SYNTHESIZED_WIRE_22 <= U3 AND SYNTHESIZED_WIRE_21;
b2v_inst7 : paralell_adder2b
PORT MAP(R0 => SYNTHESIZED_WIRE_17,
  R1 => SYNTHESIZED_WIRE_18,
  R2 => SYNTHESIZED_WIRE_19,
  R3 => SYNTHESIZED_WIRE_20,
  F0 => SYNTHESIZED_WIRE_21,
  F1 => SYNTHESIZED_WIRE_22,
  F2 => SYNTHESIZED_WIRE_22, F3
  => SYNTHESIZED_WIRE_22,
  A0 => Q0,
  A1 => Q1,
  A2 => Q2,
  A3 => Q3);
b2v_inst8 : parallel_adder1
PORT MAP(V0 => V0,
  V1 => V1,
  V2 => V2,
  V3 => V3,
  U0 => U0,
  U1 => U1,
  U2 => U2,
  U3 => U3,
  R0 => SYNTHESIZED_WIRE_17,
  R1 => SYNTHESIZED_WIRE_18,
  R2 => SYNTHESIZED_WIRE_19,
  R3 => SYNTHESIZED_WIRE_20,
  C0 => SYNTHESIZED_WIRE_5);
END
```

VHDL codes of the memoryless RNS Forward Converter

3.2.5 The RNS Based Smith - Waterman Processor

The next step after the binary/decimal conversion to RNS phase is the RNS base SWA processor stage. This stage shows how the inherent properties of RNS are used to do carry free arithmetics on the SWA. The design diagram consists of two multiplexers (MUXs), each consisting of eight inputs and four outputs, two modulus 15 parallel adder, one modulus 16 parallel adder and a control unit that controls the data selections in the two MUXs.

Each of these components are implemented using the design flow diagram as outlined in Figure 3.1 and then interconnected to get the total design. The residues produced by the forward converter are added either to the S(i, j) or (-d), where d is 2 in this design, which is the default value in literature. The Sequence of the addition is as follows:

Table 3.2: Control Unit Implementation Table and Excitation equations

Direction	Current State Q_1Q_0	Next State $Q_{1Next}Q_{0Next}$
Diagonal Addition	0 0	0 1
Upper Addition	0 1	1 0
Left Addition	1 0	0 0

Direction	Current State Q_1Q_0	Implementation D_1D_0
Diagonal Addition	0 0	0 1
Upper Addition	0 1	1 0
Left Addition	1 0	0 0

$$D_1 = Q_0$$

$$D_0 = \overline{Q_0 + Q_1}$$

H(i-1,j-1) is added to S(i, j), (this is called the Diagonal addition), H(i-1, j) is added to (- d), (this is called the Upper addition) and H(i, j - 1) is added to (-d), (this is called the Left addition).

The logic in the control unit controls the multiplexer which in turn controls the sequencing of these additions. The implementation table (using D flip - flops) for the control unit is shown in Table 3.2 with their corresponding excitation equations.

The excitation equations from implementation Table 3.2 which are used to design the control unit are: $D_0 = Q_0$ and $D_1 = Q_0+Q_1$ which are obtained from the implementation table.

The results of the forward converter are used to do carry free arithmetic operations in accordance with Equation 3.1. The residues from the diagonal entry are added to $S(i, j)$ modulus 16 and $S(i, j)$ modulus 15 at one clock cycle. At another clock cycle, the upper addition is done. Here, the two complement addition is done on the $H(i-1, j)$ with -2 modulus 16 and modulus 15. The last two complement addition is done on $H(i, j-1)$ with -2 modulus 15 and modulus 16.

In Figure 3.4 the Modulus16 block in the diagram does the modulus 16 addition of the data, the Modulu15 block does the modulus 15 addition and MUX-A and MUX-B do the data selections. The residues from the forward conversion process are added to the results *Z01, Z11, Z22, Z33* of MUX-B of the Modulus16 adder. The results *G1, G2, G3, G4* without the carry bit are the residues of the binary number with respect to modulus 16. Also the residues of the modulus 15 addition is obtained by first adding the residues values from the forward converter to the results Z0, Z1, Z2,Z3 of the MUX-A by the ParallelAdder. The results of the ParallelAdder are fed into the Modulus15 adder which finally gives the residue values of the number as A1, A2, A3, A4 as shown in Figure 3.4.

These results are also loaded into a register ready for onward processing. After the design of the RNS - SWA Processor, it is compiled and then functional and timing simulation is done. The functional simulation is done to confirm that the logic functions as expected and also to verify that the correct outputs are produced for a specified set of inputs. The waveform editor within the software was used to perform this. The timing simulation is done to verify that the circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the hardware device. The functional, circuit resource utilization and timing simulation results are posted in Chapter 4 under

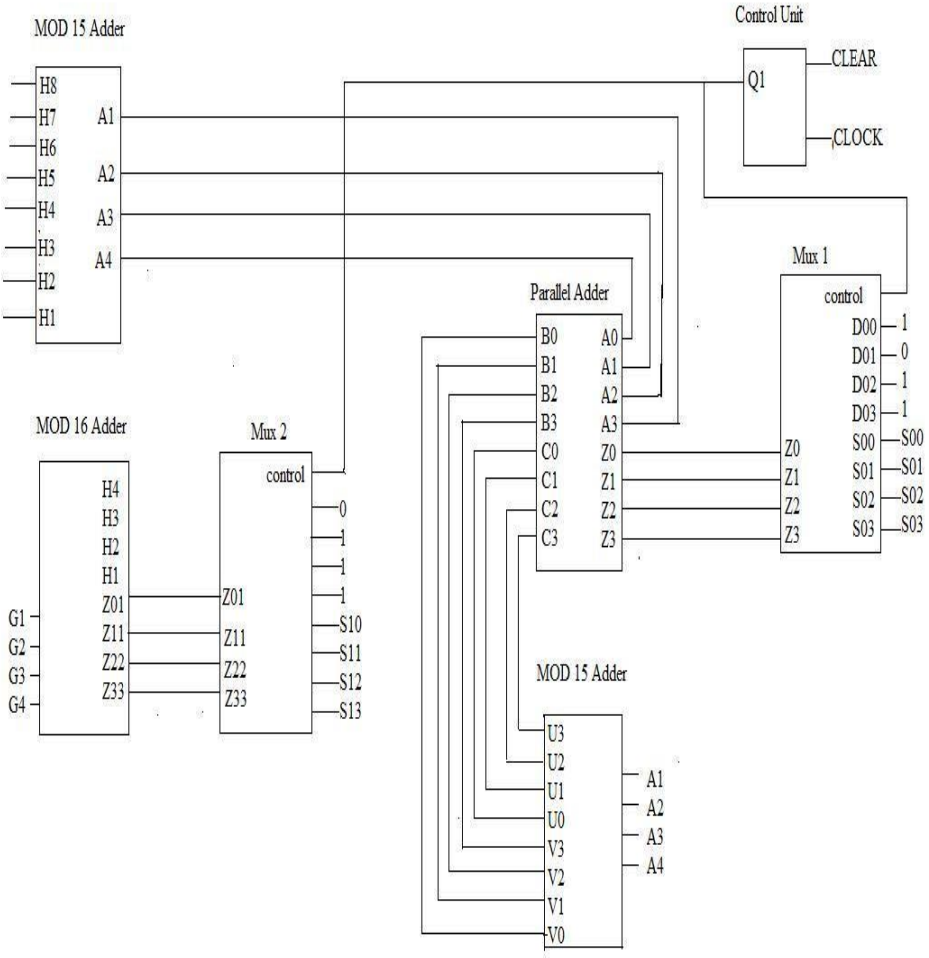


Figure 3.4: The Schematic Diagram of RNS - SWA processor

subsection 4.1.2.

3.2.6Residue Number System Comparator

The third component in the RNS - SWA architecture is the RNS Comparator. The RNS Comparator finds the maximum of the H(i, j) which consists of the

0, H(i - 1, j - 1) + S(i, j), H(i - 1, j) -d, H(i, j - 1) - d entries

The schematic diagram of the RNS - SWA comparator is shown in Figure 3.5. It compares the residues values of 0, H(i - 1, j - 1)+ S(i, j), H(i-1, j) and H(i, j - 1) to obtain the maximum value which is then assigned to H(i, j) as the matrix score.

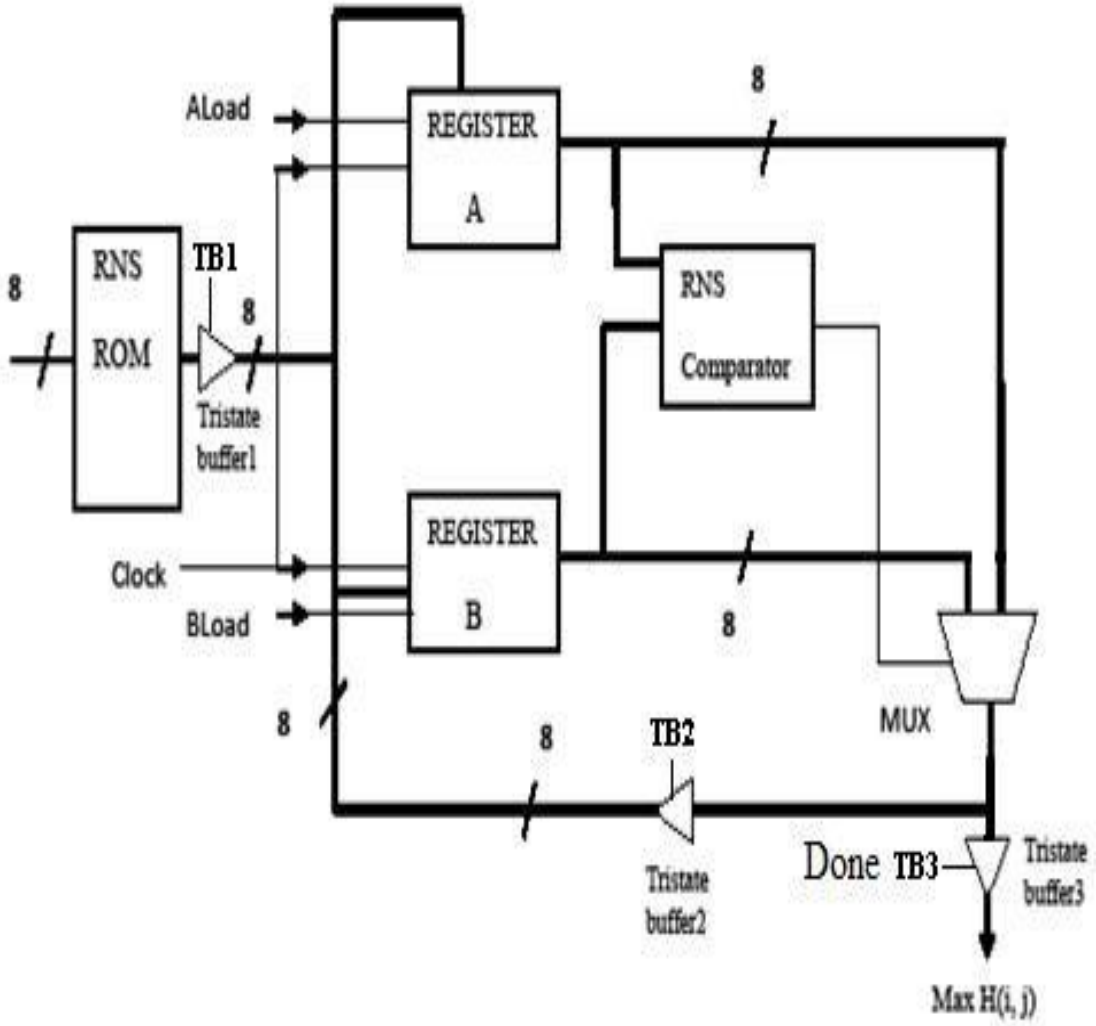


Figure 3.5: The Schematic Diagram of RNS Comparator The RNS comparator architecture is made up of 256 words × 8 - bit ROM that contains the residue values and their decimal equivalents of all the decimal numbers within the dynamic range. The residue values, 8 - bit numbers, are used as the address of their various equivalent decimal numbers. Figure below shows the VHDL codes of the ROM
and its content.

--This is a 256 X 8 bits ROM that stores the RNS

```
--values at various address locations Library
ieee; use ieee.std_logic_1164.all; use
ieee.std_logic_arith.all; use
ieee.std_logic_unsigned.all;
----------------------------------------------------------entity RNS_ROM2 is port(
Clock   : in std_logic; Clear : in std_logic;
Enable    : in std_logic;
Read : in std_logic;
A0,A1,A2,A3,A4,A5,A6,A7 : in std_logic; Data_out : out
std_logic_vector(7 downto 0)); end RNS_ROM2;
----------------------------------------------------------------architecture Behav of
RNS_ROM2 is type RNS_ROM2_Array is array (0 to 255) of
std_logic_vector(7 downto 0); constant Content: RNS_ROM2_Array := (
-- The ROM table representation of the RNS Values
0=>"00000000",
1=>"00010000",
2=>"00100000",
3=>"00110000", ...


250=>"00000000",
251=>"00000000",
252=>"00000000",
253=>"00000000",
254=>"00000000",
     OTHERS => "ZZZZZZZZ"
); signal Index:std_logic_vector(7 downto 0); begin process(Clock, Clear,
Read, Index) begin
            if( Clear = '1' ) then
      Data_out <= "ZZZZZZZZ"; elsif( Clock'event and Clock = '1' ) then if Enable
= '1' then if( Read = '1' ) then
Index<=(A0&A1&A2&A3&A4&A5&A6&A7);
```

```
Data_out <= Content(conv_integer(Index)); else
                Data_out <= "ZZZZZZZZ"; end if; end if; end if;
end process; end Behav;
```

These decimal numbers are read into two different registers at various clock cycles and then compared by the RNS comparator.The RNS Comparator does two comparison, i.e H(i - 1, j - 1) + S(i, j) with H(i - 1, j) -d to get the maximum value. And then this maximum value is compared with H(i, j-1) -d to obtain the overall maximum value. The value 0 is read from the ROM into the Register only when any of these three H(i,j)'s entries are negative since the algorithm does not deal with negative values.

In this implementation, the H(i - 1, j - 1) + S(i, j) value is loaded into Register A by asserting the Load control signal of the Register A (i.e., asserting LDA) at the first clock cycle. The actual storing of the value in Register A occurs at the beginning of the next active edge of the clock. In the next clock cycle, the H(i, j - 1) - d value is loaded into Register B by asserting the Load control signal of the register (i.e., asserting LDB) and these two register contents are compared by the RNS Comparator.

The maximum of this first comparison is selected by the multiplexer (MUX) and loaded into Register A, by asserting the LDA of the register the second time. Then the last value of H, H(i - 1, j) - d, is loaded into Register B and the second comparison is made between the two register contents, and then output the final comparison result as the Max H(i, j), by asserting the TB3 which is also connected to the Done signal. The Done control signal is to notify the external world that the execution of the algorithm has been completed and that the data at the Data output is valid. Various tristate buffers namely TB1, TB2, TB3 are asserted at appropriate times to control the data movement within the datapaths. The RNS Comparator implementation table with the corresponding excitation equations are shown in Table 3.3 The next state table is shown in Table 3.4. Since there is a total of eight states, three flip-flops are needed to encode them. For simplicity, the straight binary encoding scheme is used for encoding the states. In the next state table, these eight states are assigned to eight rows, each labeled with the state name and their encoding. In addition to the eight current states listed down the rows of the table, the next state of the FSM is also depended on the

status signal for the test condition(i.e. Start = 0) for when the condition is false and one column with the label (Start = 1)for when the condition is true. The three flip-flops

Table 3.3: Control and Output Signals of the RNS Comparator

State $Q_3Q_1Q_0$	TB1	TB2	TB3	LDA	LDB	Done
0 0 0	0	0	0	0	0	0
0 0 1	1	0	0	1	0	0
0 1 0	1	0	0	0	0	0
0 1 1	0	1	0	1	0	0
1 0 0	1	0	0	0	1	0
1 0 1	0	1	0	1	0	0
1 1 0	0	0	1	0	0	1

and one status signal give us a total of four variables (or 2^4 different combinations) to consider in the next-state table. Each next - state entry in the table is obtained from the state diagram by looking at the corresponding current state and the edges leading out from that state to see what the next is.

From the next -state table, we get the implementation table, as shown in Table 3.4. Using D flip-flops to implement the FSM, the implementation is the same as the next state table because the characteristic equation for the D flip-flop is $Q_{next} = D$. The only difference between the two tables is that the bits in the entries mean something different. In the next-state table, the bits in the entries (labeled Q_1nextQ_0next)are the next states from the FSM to go to. In the implementation table, the bits (labeled $D_2D_1D_0$) are the inputs necessary to realize those next states.

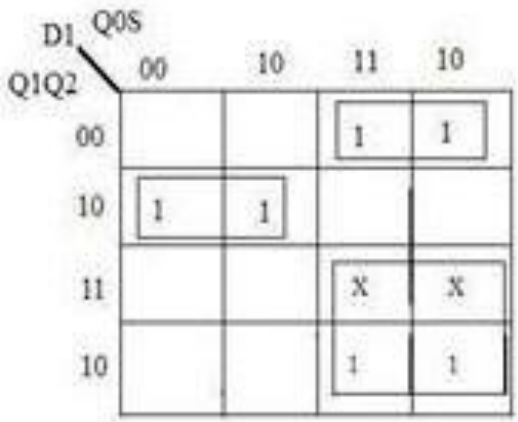
From the implementation table, we derive the excitation equations. The excitation equations are used to derive the next-state circuit for generating the inputs to the state memory flip-flops. Since we have used three D flip-flops, three excitation equations (one for D_2, one for D_1, and one for D_0) are needed, as shown in Table 3.4. The two K - maps for these two excitation equations are obtained from extracting the corresponding bits from the implementation table. These three excitation equations are depended on the three variables, Q_1,Q_0, and Start, which represent the current state and status signal respectively. Having derived the excitation equations, it is trivial to draw the next -state circuit based on these equations using the K - map method 3.5.

Table 3.4: The Next State and Implementation Tables of the Control Unit

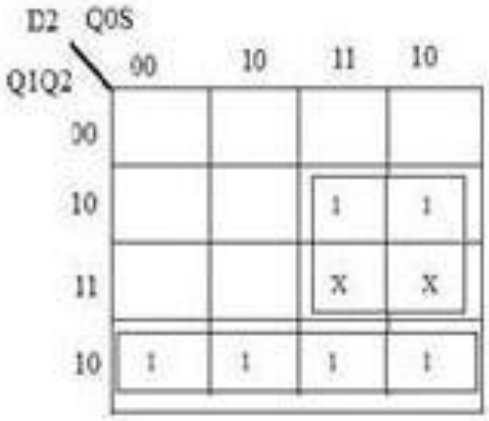
CURRENT STATE	NEXT STATE Q2NextQ1NextQ0Next	
Q2Q1Q0	Start = 0	Start = 1
000	000	000
001	010	010
010	011	011
011	100	100
100	101	101
101	110	110
110	111	000
111	XXX	XXX

CURRENT STATE	IMPLEMENTATION D2D1D0	
Q2Q1Q0	Start = 0	Start = 1
000	000	000
001	010	010
010	011	011
011	100	100
100	101	101
101	110	110
110	111	000
111	XXX	XXX

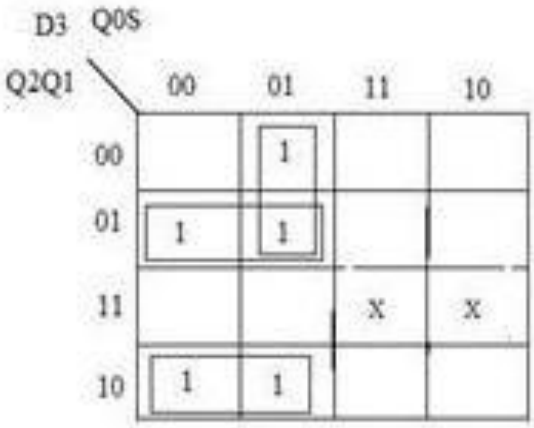
Table 3.5: The K - Map and Excitation Equations the Control Unit



$D1 = \overline{Q1}Q0 + \overline{Q2}Q1\overline{Q0} + Q2Q0$



$D2 = Q2\overline{Q1} + Q1Q0$



$D3 = (Start + Q2 + Q1)(\overline{Q2} + \overline{Q1})(\overline{Q0})$

The output logic circuit for the FSM is derived from the control word signals and the states in which the control words are assigned to. Recall that the control signals control the operation of the datapath, and now we are constructing the control unit to control the datapath. So what the control unit needs to do is to generate and output the appropriate control signals in each state to execute the instruction that is assigned to that state. In order words, the control signals for controlling the operation of the datapath are simply the output signals from the output logic circuit in the FSM. Once we have derived the excitation and output equations, we simply can draw the control unit circuit. The state memory simply consists of two D - flip-flops with asynchronous clear signals. All the asynchronous clear signals are connected to the global Reset signal. Both the next-state logic circuit and the output logic circuit are combinational

circuits and are constructed from the excitation equations and output equations respectively. The control signal and the output equations from the Table 3.3 are shown below:

$$TB1 = \overline{Q_2}\,\overline{Q_1}Q_0 + Q_2\overline{Q_1}\,\overline{Q_0} + Q_2Q_1\,\overline{Q_0}$$

$$TB2 = \overline{Q_2}Q_1Q_0 + Q_2\,\overline{Q_1}\overline{Q_0}$$

$$TB3 = Q_2Q_1\overline{Q_0}$$

$$LDA = \overline{Q_1}Q_0 + Q_0$$

$$LDB = Q_2\overline{Q_1}\,\overline{Q_0}$$

$$Done = Q_2Q_1\overline{Q_0}$$

3.2.7The Implementation Strategy of the RNS - SWA Comparator

The RNS - SWA comparator is entered into a *Quartus II version 4.0 VHDL* application software using the graphic entry or schematic capture tool embedded in the software.

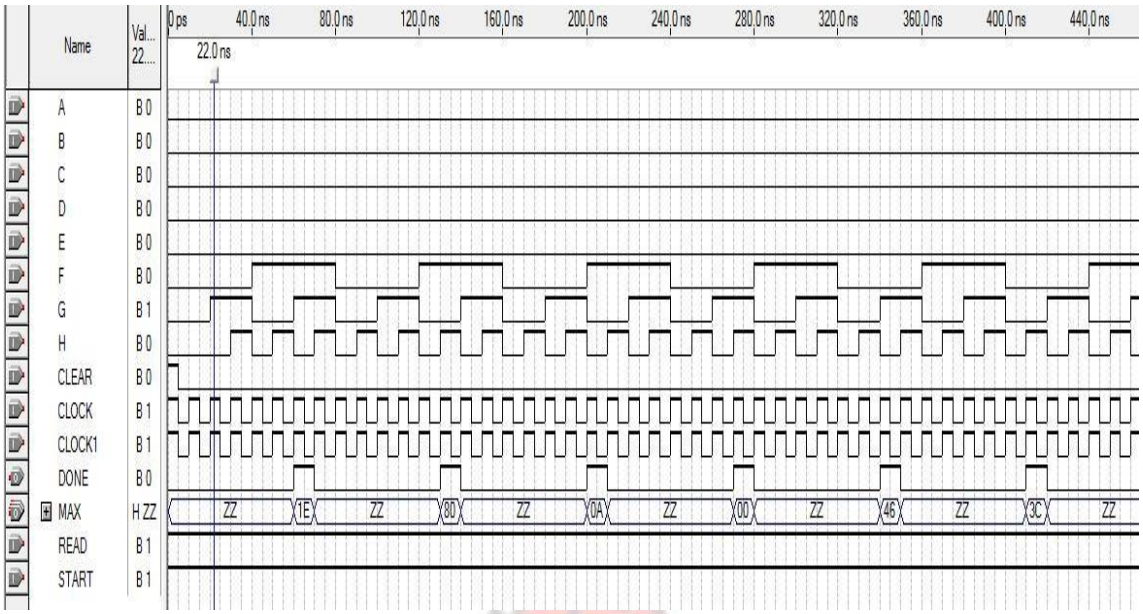


Figure 3.6: Simulation results of the RNS Comparator

This process allows pre-stored logic functions from the software library to be selected, placed on the screen, and then interconnected to create the logic design.

After the design entry is completed, it is compiled, in order to translate the source object code into object code in format that can be logically tested or downloaded to a target device.

The next step that follows the compilation process is the functional simulation. This is done by the software to confirm that the logic circuit functions as expected. The simulation will verify that correct outputs are produced for a specified set of inputs, and it is the waveform editor (a device independent software tool) that is used to verify this.

Finally, timing simulation was done to verify that the circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the hardware device.

Figure 3.6 shows the simulation waveform results of the RNS - SWA comparator.

The hardware resource utilization of the implementation are shown in Table 3.6. Form the table it is clear that the implementation is not hardware intensive as only 187 out of 10,570, making a negligible 1 of the logic cell within the device are used when

Table 3.6: The RNS - SWA Comparator Simulation status and circuit resource utilization table

Resource	Usage
Logic cells	187 / 10,570 (1 %)
Registers	36 / 12,506 (< 1 %)
Total LABs	27 / 1,057 (2 %)
Logic cells in carry chains	8
User inserted logic cells	0
I/O pins	24 / 336 (7 %)
– Clock pins	5 / 16 (31 %)
Global signals	3
M512s	0 / 94 (0 %)
M4Ks	0 / 60 (0 %)
M-RAMs	0 / 1 (0 %)
Total memory bits	0 / 920,448 (0 %)
Total RAM block bits	0 / 920,448 (0 %)
DSP block 9-bit elements	0 / 48 (0 %)
Global clocks	3 / 16 (18 %)
Regional clocks	0 / 16 (0 %)
Fast regional clocks	0 / 8 (0 %)
DIFFIOCLKs	0 / 16 (0 %)
SERDES transmitters	0 / 44 (0 %)
SERDES receivers	0 / 44 (0 %)
Maximum fan-out node	SWA_COMPARATOR_A:inst|RNS_ROM2:inst10|Index[6]
Maximum fan-out	44
Total fan-out	741
Average fan-out	3.50

implemented on EP1S10F484C5 device (a Stratix family). Also the worst - case clock -to -output delay (tco) between the specified source and destination points is 3.715 ns. These results show that the implementation is both speed and hardware efficient and will eventually improve the overall RNS - SWA architecture, thereby reducing the computational cost associated with the algorithm.

Chapter 4

Simulation Results and Discussion

This chapter discusses the simulation results of the hardware implementation of the RNS - SWA architecture.

After describing a digital system in VHDL either behaviorally or schematically, simulation of the VHDL code is important for two reasons. First, we need to verify that the VHDL code correctly implements the intended design, and second, we need to verify that the designs meets its specifications. Before the VHDL model of a digital system can be simulated, the VHDL code must first be compiled. The VHDL compiler, also called an analyzer, first checks the VHDL source code or schematic interconnections to see that it conforms to the syntax and semantic rules of VHDL. The compiler also checks to see that references to libraries are correct. If the VHDL code or schematics conforms to all of the rules, the compiler generates intermediate code which can be used by simulator or by a synthesizer.

There are basically two types of simulations used in VHDL; Functional Simulation and Timing Simulation. The functional simulation is done by the software to confirm that the logic circuit functions as expected and the timing simulation is done to verify that the circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the hardware device.

4.1Simulation Results

As outlined in section 3.2.1,the complete implementation of the RNS - SWA architecture involves basically three components. These components include:

- The Binary/Decimal to RNS Conversion stage. This step is christened the RNS Based SWA Forward converter (RSFC).

- The RNS based arithmetic operations stage. This is also termed the RNS base SWA microprocessor stage and,

- The RNS comparison stage.

The simulation results of the various components and then the complete unit of the RNS - SWA architecture are explained in the following subsections.

4.1.1The Simulation Results of the RNS Forward converter

After the schematic design entry of the memoryless RNS (mRNS) forward converter of the SWA architecture is completed, it is compiled, in order to translate the source object code into object code in format that can be logically tested or downloaded to a target device.

The next step that follows the compilation process is the functional simulation. The simulation will verify that correct outputs are produced for a specified set of inputs, and it is the waveform editor (a device independent software tool) that is used to verify this.

Finally, timing simulation was done to verify that the mRNS forward converter circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the target hardware device. The simulation result of the mRNS forward converter are shown in Figure 4.1.

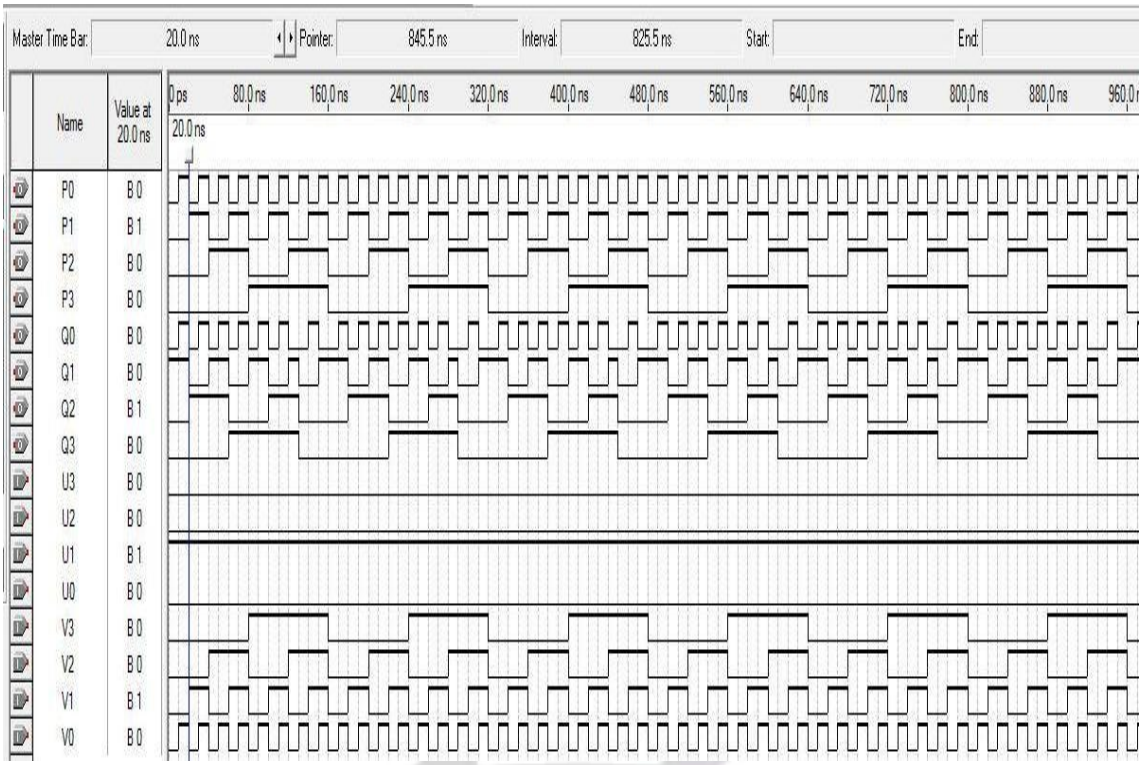


Figure 4.1: Simulation result of the mRNS Forward Converter

The hardware timing and resource utilization summary of the mRNS Forward Converter implementation are shown in Table 4.1 and Table 4.2 respectively. From the

Table 4.1: Timing Results of the mRNS Forward converter

Type	Slacks	Required Time	Actual Time	From	To
Worst-case minimum tpd	N/A	None	4.600 ns	V0	P0
Worst-case tpd	N/A	None	9.100 ns	U3	Q1

tables it is clear that the implementation is not hardware intensive as only 62% of the logic cell within the device are used. Also The worst - case point -to - point delay (tpd) between the specified source and destination points is 4.600ns making this converter both area and speed efficient.

4.1.2 The Simulation Results of RNS - SWA microprocessor

After the RNS - SWA processor is designed and entered schematically into the design software, it is compiled and simulated. The functional simulation as explained in the previous subsection is done to confirm that the logic circuit functions as expected

Table 4.2: The mRNS Simulation status and circuit resource utilization table

Flow Status	Successful - Tue Sep 07 00:00:49 2010
Revision Name	MODULUS15_16
Top-level Entity Name	MODULUS15_16
Family	MAX7000AE
Total macrocells	20 / 32 (62 %)
Total macrocells	20 / 32 (62 %)
Total pins	20 / 36 (55 %)
Device	EPM7032AELC44-4
User inserted logic cells	0
Shareable expanders	12 / 32 (37 %)
Registers used	0 / 32 (0 %)
Parallel expanders	9 / 30 (30 %)
Number of pterms used	87
Maximum fan-out node	V0
Maximum fan-out	18
Logic cells	20 / 32 (62 %)
I/O pins	20 / 36 (55 %)
Global signals	0
Cells using turbo bit	20 / 32 (62 %)
Average fan-out	3.44
– Dedicated input pins	0 / 2 (0 %)
– Clock pins	0 / 2 (0 %)

and the timing simulation was done also to verify that the circuit works at the design frequency and that there are no propagation delays or other timing problems that will affect the overall operation of the circuit when implemented on the hardware device.

The simulation of the RNS - SWA processor are shown in Figure 4.2.

Table 4.4 shows the flow summary of the RNS - SWA processor and the hardware utilization figures. Less than 1% of logic cells are employed in this implementation, making the implementation hardware efficient.

4.1.3The Simulation Results of the RNS Comparator

The schematic of RNS - SWA comparator is entered into a *Quartus II version 4.0 VHDL* application software using the graphic entry or schematic capture tool embedded in the software. This process allows pre-stored logic functions from the software library to be selected, placed on the screen, and then interconnected to create the logic

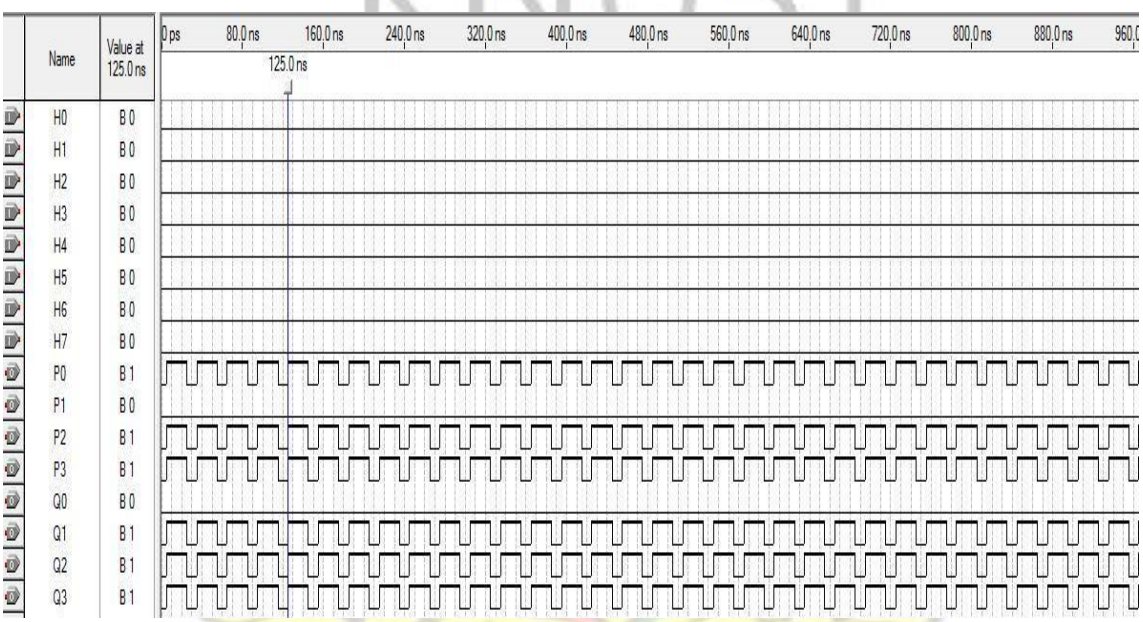


Figure 4.2: Simulation results of the RNS - SWA Processor

Flow Status	Successful - Tue Nov 23 23:57:09 2010
Revision Name	RNS_PROCESSOR
Top-level Entity Name	RNS_PROCESSOR
Family	Stratix II
Total combinational functions	27
Total registers	10
Total pins	26 / 343 (7%)
Total memory bits	0 / 419,328 (0 %)
DSP block 9-bit elements	0 / 96 (0 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)
Device	EP2S15F484C3

Total ALUTs	34 / 12,480 (1 %)

Table 4.3:

The RNS - SWA processor Simulation status and circuit resource utilization table design.

After the design entry is completed, it is compiled, in order to translate the RNS SWA comparator source code into object code in format that can be logically tested or downloaded to a target device.

What follows the compilation stage is the functional and timing simulation. These are done to check the circuit functionalities and timing constraints of the design in the target device. Figure 4.3 shows the simulation results of the RNS - SWA comparator.

The hardware resource utilization of the implementation are shown in Table 4.4

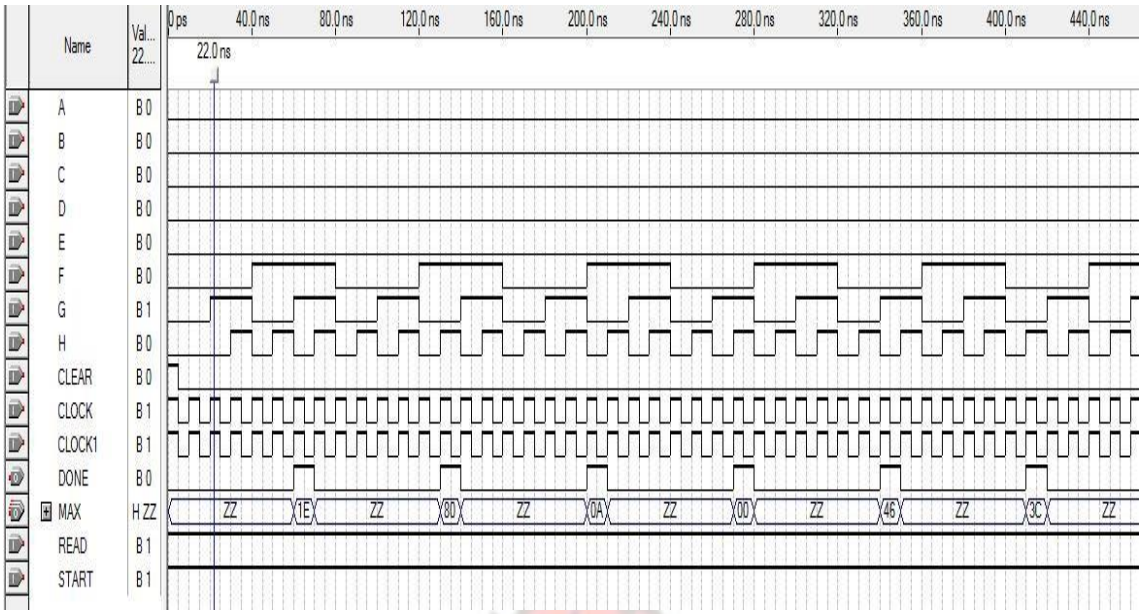


Figure 4.3: Simulation results of the RNS Comparator

Form the table it is clear that the implementation is not hardware intensive as only 187 logic cell within the device are used when implemented on EP1S10F484C5 device (a Stratix family). Also the timing results show that the worst - case clock -to -output delay (tco) between the specified source and destination points is 3.715 ns. These results show that the implementation is both speed and hardware efficient and will eventually improve the overall RNS - SWA architecture, thereby reducing the computational cost associated with the algorithm.

4.1.4The Simulation Results of the Complete RNS - SWA Architecture

The complete design of the RNS - SWA architecture is obtained when all the three stages outline above are interconnected to form a single unit using the schematic capture tool embedded in the VHDL software. This was done and functional and timing simulations carryout again on this single unit to get the overall resource utilization and delay within the design.

Table 4.4: The RNS - SWA Comparator Simulation status and circuit resource utilization table

Resource	Usage

Logic cells	187 / 10,570 (1 %)
Registers	36 / 12,506 (< 1 %)
Total LABs	27 / 1,057 (2 %)
Logic cells in carry chains	8
User inserted logic cells	0
I/O pins	24 / 336 (7 %)
– Clock pins	5 / 16 (31 %)
Global signals	3
M512s	0 / 94 (0 %)
M4Ks	0 / 60 (0 %)
M-RAMs	0 / 1 (0 %)
Total memory bits	0 / 920,448 (0 %)
Total RAM block bits	0 / 920,448 (0 %)
DSP block 9-bit elements	0 / 48 (0 %)
Global clocks	3 / 16 (18 %)
Regional clocks	0 / 16 (0 %)
Fast regional clocks	0 / 8 (0 %)
DIFFIOCLKs	0 / 16 (0 %)
SERDES transmitters	0 / 44 (0 %)
SERDES receivers	0 / 44 (0 %)
Maximum fan-out node	SWA_COMPARATOR_A:inst|RNS_ROM2:inst10|Index[6]
Maximum fan-out	44
Total fan-out	741
Average fan-out	3.50

Table 4.5 shows the summary results of the final implementation. The hardware timing and resource utilization of the implementation shows that the implementation is fast and efficient. For only 189 out of 12,480, making a negligible 1% of the logic cells within the device are used when implemented on EP2S15F484C3 device (a Stratix family). Also the worst - case clock -to -output delay (tco) between the specified source and destination points is 6.006 ns, a maximum clock speed of 185.53 MHz.

The VHDL codes for the complete RNS - SWA design are posted at Appendix A.

4.1.5 Performance Evaluation of the RNS - SWA Processor

In order to evaluate the performance of the design, it was compared with the work of Laiq Hasan and Zaid Al - Ars. Their work was chosen because of implementaTable 4.5: Flow summary and circuit resource utilization table of the RNS - SWA Architecture

Flow Status	Successful - Mon Nov 29 13:48:01 2010
Revision Name	FINAL_SWA_PROCESSOR
Top-level Entity Name	FINAL_SWA_PROCESSOR
Family	Stratix II
Total combinational functions	143
Total registers	46
Total pins	32 / 343 (9 %)
Total memory bits	0 / 419,328 (0%)
DSP block 9-bit elements	0 / 96 (0 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)
Device	EP2S15F484C3
Total ALUTs	189 / 12,480 (1 %)
Device	EP2S15F484C3
Total ALUTs	189 / 12,480 (1 %)

tion similarity and the facts that their work is currently claimed by them as the best implementation of the SWA.

In [1], Laiq Hasan and Zaid Al - Ars in 2007, used the GNU profiler, gprof, to profile the SWA in order to get the function that consumes most of the computation time. Table 4.6 shows the profiling results that was obtained. The GNU profiler gives information about the number of times, each function is called and the number of Clocks Ticks consumed by each function. The code was run on the Intel Pentium - IV (3.2

GHz) processor, for which the time period of the clock is

$$= \frac{1}{3 \cdot 2 GHz} = 0 \cdot 312 ns$$

The matrix fill function, labeled "fill_matrix_2 "in Table 4.6 was identify as the the most called function and consumed 72.33% of the total runtime, making it the right candidate to be implemented in hardware. In the table, the fill_matrix_2 function took 5.23ms of the total time. This is actually the time when the code is repeated 100 times. So the actual time consumed by the matrix fill stage function is

$$\frac{5{\cdot}23}{100} ms = 0{\cdot}05232 ms$$

Also in Laiq Hasan and Zaid Al - Ars[1] 2007, the post place and route simulation Table 4.6: L. Hasan and Z. Al-rs Profiling Results for the Software implementation of the SWA

Function	No. of Calls	No. of Clocks Ticks	No. of Clock Cycles	Total time (ms)	% Time
Init_Matrix	100	71944	2302208	0.718	9.93
fill_Matrix_1	100	32392	1036544	0.323	4.47
fill_Matrix_2	4800	524040	16769280	5.23	72.33
trace_back_1	100	31232	999424	0.312	4.31
trace_back$_2$	500	64944	2078208	0.648	8.96

showed that the total delay of their hardware implementation was $0{\cdot}0146\mu s$, whereas the time consumed by it's software equivalent was $52{\cdot}32\mu s$. Their runtime improvement over that of the software implementation was calculated to be 3582% = 35.82 times faster. The device utilization summary shows that 29 out of 13696 slices are used, so the design was claimed to be very efficient in terms of resource utilization.

In our implementation, the same matrix fill stage, labeled as "fill_matrix_2 "in the table 4.6 was implemented in hardware using RNS as a tool to further improves upon the computational cost associated with the fill_matrix_2. The VHDL implementation was run on the Intel(R) Pentium(R) Dual CPU T2330 1.60 GHz(2 CPUs) processor, for which the time period of the clock is

$$= \frac{1}{1{\cdot}6GHz} = 0{\cdot}625ns$$

The flow summary of the implementation of the RNS - SWA is shown in Table 4.5. The timing simulation of the RNS - SWA architecture shows that the critical delay is equal to 6.006ns.

The comparison between the total delay of [1] hardware implementation, denoted as Hardward1_runtime, and the hardware implementation of our work, denoted as Hardware2_runtime, expressed as a percentage, will give us the percentage runtime improvement over their work and thereby gives us a good ground to argue. Mathematically, the percentage runtime improvement ratio of the RNS - SWA implementation to that of [1] is calculated as follows:

$$\%\text{Runtime Ratio} = \frac{\frac{1}{Hardware2\_runtime}}{\frac{1}{Hardware1\_runtime}} * 100\%$$

Substituting our hardware implementation total delay denoted as Hardware2_runtime and that of [1], denoted as Hardware1_runtime, we obtain;

$$\%\text{Runtime Ratio} = \left[\frac{14{\cdot}6X10^{-9}}{6{\cdot}006X10^{-9}}\right] * 100\%$$

$$\%\text{Runtime Ratio} = 243\%$$

It will also be of interest to find the percentage difference gained in term of speed of our implementation over that of [1]. This percentage gain is calculated mathematically below:

$$\%\text{Runtime Improvement over Hardware1} = \frac{Hardware12\_runtime - Hardware11\_runtime}{\frac{1}{Hardware1\_runtime}} * 100\%$$

Substituting our hardware implementation total delay denoted as Hardware2_runtime and that of [1], denoted as Hardware1_runtime, we obtain;

$$\%\text{Runtime Improvement over Hardware1} = \left[\frac{\frac{1}{6{\cdot}006X10^{-9}} - \frac{1}{14{\cdot}6X10^{-9}}}{\frac{1}{1{\cdot}46X10^{-8}}}\right] * 100\%$$

$$\%\text{Runtime Improvement over Hardware1} = 143{\cdot}09\%$$

For the purpose of completeness and to be able to show the superiority of our work over the software implementation, our hardware implementation

result is also compared with the total time consumed by it's software equivalent as shown below:

$$\%\text{Runtime Improvement over Software} = \left[\frac{\frac{1}{Hardware2\_runtime} - \frac{1}{Software1\_runtime}}{\frac{1}{Software\_runtime}}\right] * 100\%$$

where Software_runtime is the software implementation delay. Substituting our hardware implementation total delay and that of the software's delay, we obtain;

$$\%\text{Runtime Improvement over Software} = \left[\frac{\frac{1}{6{\cdot}006X10^{-9}} - \frac{1}{52{\cdot}32X10^{-6}}}{\frac{1}{52{\cdot}32X10^{-6}}}\right] * 100\%$$

%Runtime Improvement over Software = 871029%

%Runtime Improvement over Software = 8710·29 times

From the above three comparison, these deductions can be made:

- Our hardware implementation is 243% better in term of speed over that of [1].
- In term of percentage difference gained in term of speed, our implementation is 143% superior over that of [1].
- In comparing our implementation with it's software profiling results, ours is 871029% better in term of speed, that is 8710.29 times faster.

What these outstanding results mean is that there is hope for the bioinformatics community so far as accurate sequence alignment is concern. The total time that will be needed to alignment two strings of DNA using the RNS - SWA architecture will be improved by 8710.29 times more than it's software equivalent implementation or 243% faster than that of the hardware implementation done by [1].

These results also support the fact that RNS is a good platform to implement the SWA, since it has a high prospect of improving the overall computational cost and the hardware foot print of the algorithm.

Chapter 5

Conclusion and Future Research Directions

5.1 Conclusion

This thesis work investigated the possibility of accelerating the Smith - Waterman algorithm (SWA) using the arithmetic advantages of the Residue Number System (RNS). RNS is such an integer system exhibiting the capabilities that support parallel computation, carry free addition, borrow-free subtraction, and single step multiplication without partial product. The theoretical analysis shows the advantages of implementing the SWA on an RNS platform. These advantages are exploited in this implementation to build an RNS - SWA architecture in order to reduce the computational time of the SWA. The RNS - SWA architecture consists of a Binary to RNS converter, two RNS processors, and RNS to Binary converter cum comparator.

As most existing devices and applications use binary representations, such as fixed point or floating - point numbers, the first part of an RNS design is usually a converter that converts binary numbers into residues format, which is usually termed as the forward converter or residue generator. Due to this, a customized memoryless RNS forward converter was implemented to convert the input binary representation into their respective residues forms. The RNS processors then use these residues to do

fast arithmetic operations in accordance with the SWA arithmetic operations. Various control units are built to control the sequencing of these arithmetic operations and the asserting of control signals and status signals at appropriate times.

The results obtained from the processors were converted back to their equivalent binary values by the use of 256 words × 8 - bits ROM and then comparison was done to get the maximum matrix score. The VHDL implementation of the RNS - SWA architecture shows that our implementation is superior in term of speed as compare to [1]. The runtime ratio of our implementation to that of [1] expressed as a percentage shows a 243% improvement. In terms of comparison with the hardware implementation done by [1], our hardware implementation is superior by 143%.

Also, comparing our hardware implementation result with it's software equivalent shows a tremendous improvement of 871029%, that is 8710.29 times faster. These results also support the fact that RNS is a good platform to implement the SWA, since it has a high prospect of improving the overall computational cost and the hardware foot print of the algorithm. In terms of hardware utilization, our implementation consumed 189 logic cells when implemented on EP2S15F484C3 device (a Stratix family).

What these findings mean is that there is hope for the bioinformatics community so far as accurate sequence alignment is concern. The total time that will be needed to alignment two strings of DNA using the RNS - SWA architecture will be improved by 8710.29 times more than it's software equivalent implementation. These findings support the fact that RNS is good platform to implement the SWA, and therefore will go a long way to improve the computational constraints of the SWA.

5.2 Future Research Directions

- Since RNS is showing a high prospect in accelerating the SWA, it will be of interest to implement these finding in VLSI platform.
- The nature of the moduli set use in an RNS implementations has effect on the speed and area of that device. It is of interest to investigate the possibility of implementing the SWA using different moduli set and then select the best moduli set in terms of speed, area or both.
- It is of interest to research into the possibility of building a RNS-based SWA architecture with fault tolerant capabilities to detect and correct errors using Redundant Residue Number System (RRNS).
- It could be a good research effort to investigate whether there will be gains in terms of speed and area if the SWA were implemented using Polynomial Residue Number System (PRNS).
- It will be of research interest to find out whether there will be better results in terms of speed and area if the SWA is implemented on moduli set with a wide dynamic range that can take care of a long string of DNA without doing the divide- and -conquer approach assumed in this thesis.

Bibliography

[1] L. Hasan and Z. Al-Ars, "Performance improvement of the Smith - Waterman Algorithm," Annual workshop on circuits, systems and signal processing (ProRISC), (Veldhoven, The Netherlands), November 29 - 30 2007.

[2] F. Ahmed, "Pruning algorithm to reduce the search space of the SmithWaterman algorithm," 2005.

[3] A. Davidson, *A fast pruning algorithm for optimal sequence alignment*.

Technical Report, University of Alberta, Edmonton, Alberta, Canada.

[4] K. H. Lee, C. Yu, K. H. Kwong, and P. H. W. Leong

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "A basic local search tool.," *Molecular Biology*, vol. 215, 1990.

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, pp. 195 – 197, 1981.

[7] R. Zimmermann, "Binary Adder Architectures for cell-based VLSI and their Synthesis," *PhD Dissertation, Swiss Federal Inst. of Technology*, 1997.

[8] N. Szabo and R. Tanaka, *Residue Arithmetic and its Application to Computer Technology*. New York: MC-Graw-Hill, 1967.

[9] B. Parhami, "Generalized signed-digit number system: a unifying framework for redundant number representation," *IEEE Transactions on Computers, Vol. 39, No. 1, pp. 89-98*, 1990.

78

[10] T. Chen, "Maximal redundancy signed-digit systems," *Proceedings of IEEE Symposium on Computer Arithmetic, pp. 296-300, Urbana, June*, 1985.

[11] A. D. Booth, "A signed binary multiplication technique," *Quaterly J. Math. Appl. Math., Vol. 4, part 2, pp. 236-240*, 1951.

[12] K.D.Tocher, "Technique for multiplication and division for automatic binary computers," *Quaterly J. Math. Appl. Math., Vol. 11, part 3, pp. 364384*, 1958.

[13] J. E. Robertson, "A new class of digital division methods," *IEEE Trans. on Computers, Vol. c-7, pp. 218-222, September*, 1958.

[14] B. Parhami, *Computer Arithmetic and Hardware Designs*. New York, Oxford University press, 2000.

[15] G. Jaberipur and S. Gorgin, "An improved maximally redundant signed digit adder," *Journal of Computers and Electrical Engineering, Vol. 36, pp. 491-502*, 2010.

[16] F. Taylor, “Residue arithmetic: A tutorial with examples,” *IEEE comp. magazine*, pp. pp. 50 – 62., May 1984.

[17] H. Garner, “The Residue Number System,” *IRE Trans. on Electronic Computers, pp. 140-147,* 1959.

[18] F. Barsi and P. Maestrini, “Error correcting properties of redundant residue number systems,” *IEEE Transactions on Computer, Vol. c-22, No. 3 pp.307-315, March*, 1973.

[19] L. Yang and L. Hanzo, “Redundant Residue Number System Based Error Correction Codes,” *IEEE Vehicular Technology Conference, Vol. 3, pp.1472-1476, Atlantic, NJ, USA*, 2001.

[20] V. Goh and M. Sidiqqi, “Multiple Error Detection and Correction based on Redundant Residue Number Systems,” *IEEE Trans. on Communications, Vol. 56, No. 3, pp. 325-330,* 2008.

[21] ITRS, "International technology roadmap for semiconductors, emerging research devices," *ITRS report (Executive Summary), http://www.itrs.net/Common/2007ITRS*, 2007.

[22] T. Toivonen and J. Heikkila, "Video filtering with fermat number theoretic transforms based on residue number systems," *IEEE Trans. on Circuits and Systems for Video Tech., Vol. 16, No. 1, pp. 92-101*, 2006.

[23] W. Jenkins, "Techniques for residue-to-analog conversion for residueencoded digital filters," *IEEE Trans. on Circuits and Syst., Vol. CAS-25, pp. 555-562, July*, 1978.

[24] R. Conway and J. Nelson, "Improved RNS fir filter architectures," *IEEE Trans. on Circuits and Systems-II: Express briefs, Vol. 51, No.1, pp. 26-28, January*, 2004.

[25] W. Jenkins and B. Leon, "The use of residue number systems in the design of finite impulse response digital filters," *IEEE Trans. on circuit and systems, vol. 24, pp. 191-200*, 1977.

[26] P. Fernandez, A. Garcia, J. Ramirez, L. Parrilla, and A. Lloris, "A RNSbased matrix-vector-multiply fct architecture for dct computation," *Proceedings of 43rd IEEE Midwest Symposium on Circuits and Systems, pp.350-353, Lansing, MI, August*, 2000.

[27] P. Fernandez, A. Garcia, J. Ramirez, and A. Lloris, "Fast RNS-based 2DDCT computation on field-programmable devices," *Proceedings of the IEEE Signal Processing Systems Workshop, pp.365-373, LA, USA, October*, 2000.

[28] F. Taylor, "An RNS discrete fourier transform implementation," *IEEE Trans. Acoust. Speech, Signal Process, Vol. 38, No. 8, pp. 1386-1394*, 1990.

[29] F. Taylor and C. Huang, "A comparison of DFT algorithms using a residue arithmetic architecture," *International Journal of Computer Electronic Engineering, September*, 1982.

[30] J. Vaccaro, B. Johnson, and C. Nowacki, "A systolic discrete fourier transform using residue number systems over the ring of gaussian integers," *IEEE International Conference on Accoustics, Speech and Signal Processing, pp. 1157-1160,* 1986.

[31] C. H. Huang and F. J. Taylor, "High speed DFTs using residue numbers," *in Proc. IEEE 1980 Conf Acoust., Speech, Signal Processing, Denver, Co, pp. 238-241, April,* 1980.

[32] M. Soderstrand, W. Jenkins, G. Jullien, and F. Taylor, *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. IEEE press, Piscataway, NJ, USA, 1986.

[33] R. Merrill, "Improving digital computer performance using residue number theory," *Trans. on Electronic Computers, vol. 13, issue 2, pp. 93-101*, 1964.

[34] T. Stouraitis and V. Paliouras, "Considering the alternatives in low-power design," *IEEE Circuits and Devices Magazine, Vol. 17, issue 4, pp. 22-29*, 2001.

[35] K. Parhi, "Low-energy csmt carry generators and binary adders," *IEEE Trans. VLSI Syst., vol. 7, pp. 450-462, December,* 1999.

[36] T. Callaway and E. S. Jr., "Power-delay characteristics of cmos multipliers," *in Proc. 13th Symp. Computer Arithmetic (ARITH13), Asilomar, USA, July 1997, pp. 26-32,* 1997.

[37] P. Landman and J. Rabaey, "Architectural power analysis: The dual bit type method," *IEEE Trans. VLSI Syst., vol. 3, pp. 173-187, June,* 1995.

[38] V. Paliouras and T. Stouraitis, "Signal activity and power consumption reduction using the logarithmic number system," *in Proc. 2001 IEEE Int. Symp. Circuits and Systems (ISCAS), vol. 2, pp. 653-656*, 2001.

[39] V. Paliouras and T. Stouraitis, "Low-power properties of the logarithmic number system," *in Proc. 15th Symp. Computer Arithmetic (ARITH15)*, 2001.

[40] W. A. C. Jr., "One-hot residue coding for low delay-power product cmos design," *IEEE Trans. Circuits Syst. II, vol. 45, pp. 303-313, March*, 1998.

[41] M. Ibrahim, "Novel digital filter implementations using hybrid rns-binary arithmetic," *Signal Processing, vol. 40, no. 2-3, pp. 287-294*, 1994.

[42] A. Z. Baraniecka and G. A. Jullien, "Residue number system implementation of number theoretic transforms in complex residue rings," *IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-28, pp. 285-291, June*, 1980.

[43] M. R. Schroeder, *Number Theory in Science and Communication*. Germany: Springer-Verlag, 1984.

[44] K. Y. Lin, B. Krishna, and H. Krishna, "Rings, fields, the chinese remainder theorem and an extension," *IEEE Trans. Circuits Syst. II , vol. 41, pp. 641?655, October*, 1994.

[45] A. Madhukumar, F. Chin, and A. Premkumar, "Residue number system based multicarrier CDMA for broadband mobile communication systems," *Proceedings of 43rd IEEE Midwest Symposium on Circuits and Systems, pp.536-539, Lansing, MI, August*, 2000.

[46] A. Madhukumar and F. Chin, "Performance of a residue number system based cdma system over bursty communication channels," *Journal of Wireless Personal Communications, Springer, Netherlands, Vol. 22, No. 1, pp.89-102*, 2002.

[47] T. Shahana, R. Babita, K. Jacob, and S. Sasi, "RRNS-convolutional concatenated code for OFDM based wireless communication with direct analog-to-residue converter," *Proceedings of World Academy of Science, Engineering and Technology, Vol. 35, pp.652-659,* 2008.

[48] L. Yang and L. Hanzo, "Residue number system assisted fast frequencyhopped synchronous ultra-wideband spread-spectrum multiple-access: A design alternative to impulse radio," *IEEE Journal on Selected Areas of Communications, 20 (9), pp. 1652-1663,* 2002.

[49] L. Hasan and Z. Al-Ars, "Accurate profiling and acceleration evaluation of the Smith - Waterman Algorithm using the MOLEN platform,"

[50] P. Hogeweg, "Simulating the growth of cellular forms.," *Simulation,* vol. 31, pp. 90–36, 1978.

[51] M. Borah, R. S. Bajwa, S. Hannenhalli, and M. J. Irwin, "A SIMD solution to the sequence comparison problem on the MGAP," 1994.

[52] D. P. Lopresti, "Rapid implementation of a genetic sequence comparator using field programmable logic arrays," pp. 138–152.

[53] H. Y. Liao, M. L. Yin, and Y. Cheng., "A parallel implementation of the Smith-Waterman Algorithm for Massive Sequences Searching.," September 1-5, 2004.

[54] S. A. et al, "Bio-sequence database scanning on a GPU."

[55] A. D. B. et al, "The UCSC kestrel parallel processor," vol. 16, no. 1, pp. 80–
92.

[56] S. Margerm, "Reconfigurable computing in real world applications," *Cray Inc., FPGA and Structured ASIC,* February, 7, 2006.

[57] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processor for bio-sequence database scanning on FPGAs," 2005.

[58] J. Chiang, J. Shaw, M. Studniberg, and K. Truong, "Hardware accelerator for genomic sequence alignment," August 30 - September 3, 2006.

[59] Y. Yamaguchi, T. Maruyana, Y. Miyajima, and A. Konagaya, "High speed homology search using run-time reconfiguration," 2002.

[60] B. H. W. Yang, *A parallel Implementation of Smith - Waterman Sequence Comparison Algorithm*. Stanford University, USA, December 6, 2002.

[61] M. A. Soderstrand, W. K. Jenkins, G. Jullien, and F. J. Taylor, "Residue number system arithmetic," *Modern Applications in Digital Signal Processing*, 1986.

[62] K. Gbolagade and S. Cotofana, "A residue to binary converter for the moduli set $\{2n+2,2n+1,2n\}$," Nov., 2008.

[63] K. Gbolagade and S. Cotofana, "Residue number operands to decimal conversion for 3 - moduli set," August, 2008.

[64] Y. Wang, X. Song, M. Aboulhamid, and H. Shen:, "Adder based residue to binnary number converter for $\{2^n-1,2^n,2^n+1\}$," *IEEE Trans. on Signal processing*, vol. 50, no. 7, July 2002.

[65] Y. Wang, "New chinese remainder theorems," vol. 1 of *in proc. 32nd Asilomer Conf. signals, systems computing*, pp. pp. 165 – 171, 1998.

[66] Y. Wang, "Residue - to - Binary converters based on New Chinese Remainder Theorems," *IEEE Trans. on circuits and systems II: Analog to Digital signal processing*, vol. 47, no. 3, March, 2000.

[67] K. Gbolagade and S. Cotofana, "Generalized matrix method for efficient residue to decimal conversion," No. 3 in Proceedings of 19th IEEE Asia

Pacific Conference on Circuits and Systems (APCCAS 2008), (Macao, China), pp. 1414 – 1417, Dec., 2008.

[68] H. M. Yassine and Moore, "Improved mixed - radix conversion for residue number system architectures," vol. vol. 138 of *Proceeding IE Pt. G*, pp. 120 – 124, Feb. 1991.

[69] K. Gbolagade and S. Cotofana, "An O(n) Residue Number System to Mixed Radix Conversion," (To appear) in proceeding of the 2009 IEEE International Symposium on Circuits and Systems (ISCAS 2009), (Taiwan,
China), May 2009.

[70] M. Abdallah and A. Skavantzo, "A systematic approach for selecting practical moduli sets for residue number systems," proceedings of the 27th Southeastern Symposium on System theory, pp. 445 – 449, 1995.

[71] A. B. Premkumar, "An RNS to binary converter in a three moduli set with common factors," *IEEE Trans. on circuits and systems - II analog and digital signal proc.*, vol. 42, no. 4, pp. 98 – 301, 1995.

[72] W. Wang, M. Swamy, M. O. Ahmad, and Y. Wang, "A study of theResidue - to -Binary converters for the three - moduli set," *IEEE Trans. on circuits and systems I. Fundamental theory and Applications*, vol. 50.

[73] A. Prekumar, "Improved memoryless rns forward converter based on the periodicity of residues," *IEEE Trans. on circuits and systems - II, Express briefs*, vol. 53, no. 2, 2006.

[74] M. Abdallah and A. Skavantzos, "On multimuduli residue number systems with moduli of the forms $\{r^a, r^b - 1, r^c + 1\}$," *IEEE Trans., on circuits and systems*, vol. vol. 52, no. 7, pp. 1253 – 1266, 2005.

[75] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*. New Jersey: Tom Robbins, 2000.

[76] R. Conway and J. Nelson, “Fast converter for 3 moduli RNS using new property of CRT,” *IEEE Trans. Comput., vol. 48, pp. 852-860, August,* 1999.

Appendix A

VHDL Codes implantation of the complete RNS - SWA Architecture


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY FINAL_RNS_PROCESSOR IS port
(
H7 : IN STD_LOGIC;
H6 : IN STD_LOGIC;
H5 : IN STD_LOGIC;
H4 : IN STD_LOGIC;
H3 : IN STD_LOGIC;
H2 : IN STD_LOGIC;
H1 : IN STD_LOGIC;
H0 : IN STD_LOGIC;
S00 : IN STD_LOGIC;
S01 : IN STD_LOGIC;
S02 : IN STD_LOGIC;
S03 : IN STD_LOGIC;
S10 : IN STD_LOGIC;
S11 : IN STD_LOGIC;
S12 : IN STD_LOGIC;
S13 : IN STD_LOGIC;
READ : IN STD_LOGIC;
START : IN STD_LOGIC;
CLOCK : IN STD_LOGIC;
CLEAR : IN STD_LOGIC;
CLOCK1 : IN STD_LOGIC;
CLEAR1 : IN STD_LOGIC;
DONE : OUT STD_LOGIC;
MAX : OUT STD_LOGIC_VECTOR(7 downto 0)
);
```



```
END FINAL_RNS_PROCESSOR;

ARCHITECTURE bdf_type OF FINAL_RNS_PROCESSOR IS

component rns_processor
```

```
PORT(H7 : IN STD_LOGIC;
  H6 : IN STD_LOGIC;
  H5 : IN STD_LOGIC;
  H4 : IN STD_LOGIC;
  H3 : IN STD_LOGIC;
  H2 : IN STD_LOGIC;
  H1 : IN STD_LOGIC;
  H0 : IN STD_LOGIC;
  S00 : IN STD_LOGIC;
  S01 : IN STD_LOGIC;
  S02 : IN STD_LOGIC;
  S03 : IN STD_LOGIC;
  S10 : IN STD_LOGIC;
  S11 : IN STD_LOGIC;
  S12 : IN STD_LOGIC;
  S13 : IN STD_LOGIC;
  CLEAR : IN STD_LOGIC;
  CLOCK : IN STD_LOGIC;
  Q3 : OUT STD_LOGIC;
  Q2 : OUT STD_LOGIC;
  Q1 : OUT STD_LOGIC;
  Q0 : OUT STD_LOGIC;
  P3 : OUT STD_LOGIC;
  P2 : OUT STD_LOGIC;
  P1 : OUT STD_LOGIC; P0 : OUT STD_LOGIC
); end component;

component rns_comparator
PORT(A0 : IN STD_LOGIC;
  A1 : IN STD_LOGIC;
  A2 : IN STD_LOGIC;
  A3 : IN STD_LOGIC;
  A4 : IN STD_LOGIC;
  A5 : IN STD_LOGIC;
  A6 : IN STD_LOGIC;
  A7 : IN STD_LOGIC;
  READ : IN STD_LOGIC;
  CLOCK1 : IN STD_LOGIC;
  START : IN STD_LOGIC;
  CLOCK : IN STD_LOGIC;
  CLEAR : IN STD_LOGIC;
```



```
  DONE : OUT STD_LOGIC;
  MAX : OUT STD_LOGIC_VECTOR(7 downto 0)
); end
component;
```

```
signal SYNTHESIZED_WIRE_0 : STD_LOGIC;
signal SYNTHESIZED_WIRE_1 : STD_LOGIC;
signal SYNTHESIZED_WIRE_2 : STD_LOGIC;
signal SYNTHESIZED_WIRE_3 : STD_LOGIC;
signal SYNTHESIZED_WIRE_4 : STD_LOGIC;
signal SYNTHESIZED_WIRE_5 : STD_LOGIC;
signal SYNTHESIZED_WIRE_6 : STD_LOGIC;
signal SYNTHESIZED_WIRE_7 : STD_LOGIC;
BEGIN


b2v_inst : rns_processor
PORT MAP(H7 => H7,
  H6 => H6,
  H5 => H5,
  H4 => H4,
  H3 => H3,
  H2 => H2,
  H1 => H1,
  H0 => H0,
  S00 => S00,
  S01 => S01,
  S02 => S02,
  S03 => S03,
  S10 => S10,
  S11 => S11,
  S12 => S12,
  S13 => S13,
  CLEAR => CLEAR1,
  CLOCK => CLOCK1,
  Q3 => SYNTHESIZED_WIRE_0,
  Q2 => SYNTHESIZED_WIRE_1,
  Q1 => SYNTHESIZED_WIRE_2,
  Q0 => SYNTHESIZED_WIRE_3,
  P3 => SYNTHESIZED_WIRE_4,
  P2 => SYNTHESIZED_WIRE_5,
  P1 => SYNTHESIZED_WIRE_6,
  P0 => SYNTHESIZED_WIRE_7);

b2v_inst3 : rns_comparator
PORT MAP(A0 => SYNTHESIZED_WIRE_0,
  A1 => SYNTHESIZED_WIRE_1,
  A2 => SYNTHESIZED_WIRE_2,
  A3 => SYNTHESIZED_WIRE_3,
  A4 => SYNTHESIZED_WIRE_4,
  A5 => SYNTHESIZED_WIRE_5,
  A6 => SYNTHESIZED_WIRE_6,
  A7 => SYNTHESIZED_WIRE_7,
```

```
  READ => READ,
  CLOCK1 => CLOCK,
  START => START,
  CLOCK => CLOCK1,
  CLEAR => CLEAR,
  DONE => DONE, MAX => MAX);

END;
```

Appendix B

VHDL Codes implantation of the RNS - SWA Comparator


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY Comparator1 IS port (
LOAD : IN STD_LOGIC;
CLOCK : IN STD_LOGIC;
CLEAR : IN STD_LOGIC;
A  : IN STD_LOGIC_VECTOR(7 downto 0);
B  : IN STD_LOGIC_VECTOR(7 downto 0);
MAX : OUT STD_LOGIC_VECTOR(7 downto 0)
);
END Comparator1;

ARCHITECTURE bdf_type OF Comparator1 IS

component rns_reg_a
PORT(clock : IN STD_LOGIC; aclr : IN STD_LOGIC; aload :
  IN STD_LOGIC; data : IN STD_LOGIC_VECTOR(7
  downto 0); q : OUT STD_LOGIC_VECTOR(7 downto 0)
); end component;

component rns_reg_b
PORT(clock : IN STD_LOGIC; aclr : IN STD_LOGIC; aload :
  IN STD_LOGIC; data : IN STD_LOGIC_VECTOR(7
  downto 0);
```



```
  q : OUT STD_LOGIC_VECTOR(7 downto 0)
); end component;

component rns_comp
PORT(dataa : IN STD_LOGIC_VECTOR(7 downto 0); datab : IN
  STD_LOGIC_VECTOR(7 downto 0); ageb : OUT STD_LOGIC
); end component;

component rns_mux
PORT(sel : IN STD_LOGIC; data0x : IN
  STD_LOGIC_VECTOR(7 downto 0); data1x : IN
```

```
  STD_LOGIC_VECTOR(7 downto 0); result : OUT
  STD_LOGIC_VECTOR(7 downto 0)
); end component;

signal SYNTHESIZED_WIRE_5 : STD_LOGIC_VECTOR(7 downto 0); signal

SYNTHESIZED_WIRE_6 : STD_LOGIC_VECTOR(7 downto 0); signal

SYNTHESIZED_WIRE_2 : STD_LOGIC; BEGIN



b2v_inst2 : rns_reg_a
PORT MAP(clock => CLOCK, aclr =>
  CLEAR, aload => LOAD, data => A, q =>
  SYNTHESIZED_WIRE_6);

b2v_inst3 : rns_reg_b
PORT MAP(clock => CLOCK, aclr =>
  CLEAR, aload => LOAD, data => B, q =>
  SYNTHESIZED_WIRE_5);

b2v_inst4 : rns_comp
PORT MAP(dataa => SYNTHESIZED_WIRE_5, datab
  => SYNTHESIZED_WIRE_6, ageb =>
  SYNTHESIZED_WIRE_2);

b2v_inst5 : rns_mux
```



```
        PORT MAP(sel =>
          SYNTHESIZED_WIRE_2, data0x =>
          SYNTHESIZED_WIRE_6, data1x =>
          SYNTHESIZED_WIRE_5, result =>
          MAX);

        END;
```