AD NUMBER

ADB310230

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies only; Premature Dissemination; DEC 2004. Other requests shall be referred to Superintendent, Naval Postgraduate School, Code 261, Monterey, CA 93943-5000.

AUTHORITY

NPS ltr 26 Jan 2011

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

ARCHITECTURE OF AN INTEGRATED MICROELECTRONIC WARFARE SYSTEM ON A CHIP AND DESIGN OF KEY COMPONENTS

by

Brian L. Luke

December 2004

Dissertation Supervisors: Douglas J. Fouts
Phillip E. Pace

~~Distribution authorized to U.S. Government Agencies only; premature dissemination; December 2004. Other requests for this document must be referred to Superintendent, Code 261, Naval Postgraduate School, Monterey, CA 93943-5000 via the Defense Technical Information Center, 8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060-6218.~~

Approved for public release, distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

January 25, 2011

SUBJECT: Change in distribution statement for *Architecture of an Integrated Microelectronic Warfare System on a Chip and Design of Key Components* – December 2004.

1. Reference: Luke, Brian L. *Architecture of an Integrated Microelectronic Warfare System on a Chip and Design of Key Components*. Monterey, CA: Naval Postgraduate School. Department of Electrical and Computer Engineering, December 2004.
   UNCLASSIFIED [Distribution authorized to U.S. Government Agencies only; premature dissemination; December 2004].

2. Upon consultation with NPS faculty, the School has determined that this dissertation may be released to the public and that its distribution is unlimited, effective January 25, 2011.

University Librarian
Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2004	**3. REPORT TYPE AND DATES COVERED** Dissertation	
4. TITLE AND SUBTITLE: Architecture of an Integrated Microelectronic Warfare System on a Chip and Design of Key Components			**5. FUNDING NUMBERS**
6. AUTHOR(S) Luke, Brian L., LCDR USN			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT ~~Distribution authorized to U.S. Government Agencies only; premature dissemination; December 2004. Other requests for this document must be referred to Superintendent, Code 261, Naval Postgraduate School, Monterey, CA 93943-5000 via the Defense Technical Information Center, 8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060-6218.~~ Approved for public release, distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This dissertation investigates a mixed-signal, electronic warfare (EW) system-on-a-chip (SoC) design capable of synthesizing false radar returns in response to imaging radar interrogations that, when integrated into the range-Doppler processing, form an image of a false target. Detailed designs for the EW SoC components including the false target digital image synthesizer (DIS) and a novel analog to digital converter (ADC) are provided in this research. Alternative DIS architectures are presented that reduce circuit die area and power dissipation. This research also describes the theory, design, implementation, simulation, and testing of a proof-of-concept application specific integrated circuit (ASIC) providing automatic counterflow-clock pipeline skew control for the DIS. High performance ADCs are key components of mixed-signal SoCs. Design and simulation results for an 8-bit 1 GS/s robust symmetric number system (RSNS) folding ADC are presented. The gray-code properties of the RSNS make it desirable for error control and low-power ADC implementations. A complete mathematical description of the N-modulus RSNS redundancies is discovered, which results in closed-form expressions for the longest sequence of unique RSNS vectors for moduli of the form m-1, m, and m+1, as well as an efficient search algorithm for N-modulus systems at least six orders of magnitude faster than previously published results. Lastly, an N-modulus RSNS-to-binary converter design procedure and a circuit design for an 8-bit, 4-modulus 1 GS/s RSNS-to-binary converter are presented.

14. SUBJECT TERMS Folding ADC, gray-code properties, dynamic range, residue number system, robust symmetric number system, inverse synthetic aperture radar, electronic warfare, system-on-a-chip, wideband imaging radar, digital image synthesis, radar countermeasures, anti-ship capable missile, counterflow clock pipeline, automatic clock skew control			**15. NUMBER OF PAGES** 320 / **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU ~~UL~~

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

ARCHITECTURE OF AN INTEGRATED MICROELECTRONIC WARFARE SYSTEM ON A CHIP AND DESIGN OF KEY COMPONENTS

Brian L. Luke
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1992
M.S., Naval Postgraduate School, 1998

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2004

Author: _____
Brian L. Luke

Approved by:

_____
Charles W. Therrien
Professor of Electrical and
Computer Engineering

_____
Douglas J. Fouts
Professor of Electrical and
Computer Engineering
Dissertation Supervisor

_____
Herschel H. Loomis, Jr.
Professor of Electrical and
Computer Engineering

_____
Phillip E. Pace
Professor of Electrical and
Computer Engineering
Dissertation Supervisor

_____
Cynthia E. Irvine
Professor of Computer Science

Approved by: _____
John P. Powers, Chair, Department of Electrical
and Computer Engineering

Approved by: _____
Julie Filizetti, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This dissertation investigates a mixed-signal, electronic warfare (EW) system-on-a-chip (SoC) design capable of synthesizing false radar returns in response to imaging radar interrogations that, when integrated into the range-Doppler processing, form an image of a false target. Detailed designs for the EW SoC components including the false target digital image synthesizer (DIS) and a novel analog to digital converter (ADC) are provided in this research. Alternative DIS architectures are presented that reduce circuit die area and power dissipation. This research also describes the theory, design, implementation, simulation, and testing of a proof-of-concept application-specific integrated circuit (ASIC) providing automatic counterflow-clock pipeline skew control for the DIS. High performance ADCs are key components of mixed-signal SoCs. Design and simulation results for an 8-bit 1 GS/s robust symmetric number system (RSNS) folding ADC are presented. The gray-code properties of the RSNS make it desirable for error control and low-power ADC implementations. A complete mathematical description of the *N*-modulus RSNS redundancies is discovered, which results in closed-form expressions for the longest sequence of unique RSNS vectors for moduli of the form $m-1$, m, and $m+1$, as well as an efficient search algorithm for *N*-modulus systems at least six orders of magnitude faster than previously published results. Lastly, an *N*-modulus RSNS-to-binary converter design procedure and a circuit design for an 8-bit, 4-modulus 1 GS/s RSNS-to-binary converter are presented.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF SYMBOLS, ACRONYMS, AND ABBREVIATIONS

I	In phase
$\hat{M}$	longest sequence of consecutive unique RSNS vectors
P_f	fundamental period
Q	Quadrature
X_h	vector of RSNS residues at position h

ADC	Analog to Digital Converter
ASCM	Antiship Capable Missile
ASIC	Application Specific Integrated Circuit
C^2	Counter-clock
CMOS	Complementary Metal Oxide Semiconductor
COR	Center of Redundancy
CRT	Chinese Remainder Theorem
DAC	Digital to Analog Converter
DIS	Digital Image Synthesizer
DNL	Differential Non-linearity
EW	Electronic Warfare
FSM	Finite State Machine
GIS	Geographic Information System
INL	Integral Non-linearity
ISAR	Inverse Synthetic Aperture Radar

LPS	Least Positive Solution
LSB	Least Significant Bit
LUT	Look-up Table
MRS	Modulus Residue Sequence
MRSS	Modulus Residue Sub-sequence
PRP	Pair Wise Relatively Prime
RNS	Residue Number System
ROM	Read Only Memory
RSNS	Robust Symmetric Number System
SiGe	Silicon Germanium
SmartSearch	MATLAB implementation of a new $\hat{M}$ search algorithm
SNS	Symmetric Number System
SoC	System-on-a-Chip

EXECUTIVE SUMMARY

Success of naval actions in hostile environments hinges on the ability of combatants to perform under enemy fire. The current strength and technical capability of the US Navy make it unlikely that an attack would be frontal and direct. Rather, an oblique and asymmetrical attack is expected. Modern anti-ship capable missiles (ASCM) provide adversaries with a highly capable and deadly asymmetric weapon due to their relatively low cost and advanced homing capabilities. Naval forces operating in cluttered littoral environments are particularly vulnerable from land or sea. Therefore, the susceptibility of Navy surface ships to asymmetric attack is strongly dependent on their capability to defend against the ASCM threat.

Wideband imaging radars, such as the inverse synthetic aperture radar (ISAR), are a unique class of radars that provide range and bearing to a target as well as the general shape of the target. This enables an adversary to detect as well as classify a target. Consequently, an adversary using an imaging radar and a classification table can segregate high-priority targets from low priority targets or decoys. Like other types of radars, imaging radars are susceptible to noise jamming. However, noise jamming provokes an ISAR to use anti-jamming techniques or switch to alternative methods of target detection and classification. In addition, future ASCMs are also expected to use imaging seekers in order to improve aimpoint accuracy and reject decoys. A programmable multi-chip electronic warfare (EW) system currently in development has the capability to transmit synthesized false targets to an imaging radar, which when integrated by the range-Doppler processing, will generate an image which will likely be indistinguishable from true target returns. Unlike noise jamming, the false-target EW system will not alert an adversary to the ongoing deception. Consequently, the ISAR will continue to track both the real and false targets and will not switch to alternative homing methods. The circuit that generates the false images in response to an imaging radar is called the Digital Image Synthesizer (DIS).

This dissertation proposes a design for a high-performance, mixed-signal, EW system-on-a-chip (SoC) capable of producing false targets in response to wideband imaging radar interrogations. The minimum components required for the DIS EW SoC are an analog-to-digital converter (ADC), high-speed memory, the DIS signal processor, and a digital-to-analog converter (DAC). Each component in the SoC must be optimized for low-power and small die area while maintaining a high operating frequency. Detailed designs for the false target DIS and ADC are provided in this research.

The DIS is the largest single component in terms of die area in the proposed SoC. Alternative DIS architectures are presented that attempt to reduce circuit die area and power dissipation. This research also describes the theory, design, implementation, simulation, and testing of a proof-of-concept ASIC providing automatic counterflow-clock pipeline skew control for the DIS.

High performance ADCs are key components of mixed-signal SoCs. Two of the fastest ADC architectures in use today are the flash ADC and the folding ADC. The robust symmetric number system (RSNS) folding ADC offers significant advantages over the flash ADC in terms of conversion speed, power savings, and die area. The gray-code property of the RSNS makes it particularly useful in direction finding interferometer antenna architectures and electro-optic digital antennas as well as ADCs since it eliminates encoding errors common in those systems. This work presents the design of an 8-bit 1 GS/s RSNS folding ADC and RSNS-to-binary converter suitable for implementation in an EW false target digital image synthesizer SoC.

ACKNOWLEDGMENTS

The entirety of this research is dedicated to the glory of God. May it be used for His purposes and help in some small way to bring peace to the world in His name.

I wish to express my sincerest appreciation to my co-advisor, Professor Douglas Fouts. His steadfast support and exemplary leadership helped me to plough through the difficult and frustrating periods when the light at the end of the tunnel seemed dim. His enthusiasm for engineering and endless stream of ideas made this work possible. I also wish to thank my co-advisor Professor Phillip Pace. His infectious drive and constant feedback provided the motivation for much of the research in this dissertation. My respect and admiration for him is immeasurable. I want to thank my Doctoral Committee members Professors Charles Therrien, Herschel Loomis, and Cynthia Irvine for their contributions, support and inspiration. Thank you for always keeping your doors and minds open. A special thanks goes out to David Styer for his extensive assistance.

I want to convey my gratitude to the Naval Security Group for providing me the opportunity to pursue this program. I hope that the training and education acquired in my doctoral studies will be invaluable to the NSG in the years to come.

This work was supported in part by the Office of Naval Research Code 313 and the Naval Research Laboratory Code 5740. Much appreciation goes to Dr. Peter Craig, Mr. Jim Talley, and Mr. Mike Monsma for their support and encouragement.

I wish to thank my family for their unwavering support. When the demands on my time were the greatest, their patience and faith in me was the strongest. The support of my wife, Heather, continues to remind me that there is faith for darkened hours, courage in despairing nights, calm in stressful circumstances, and to always walk in the Light. I also wish to thank my parents for their love and support for a son who rarely finds his way home. Finally, I wish to thank my children, Austin and Carson, who never seemed to notice that their Dad was too busy or too tired to play. They are the reason I continue to pledge to my service to the United States military. May God bless this work and the United States of America.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Success of naval actions in hostile environments hinges on the ability of combatants to perform under enemy fire. The current strength and technical capability of the US Navy make it unlikely that an attack would be frontal and direct. Rather, an oblique and asymmetrical attack is expected. Modern anti-ship capable missiles (ASCM) provide adversaries with a highly capable and deadly asymmetric weapon due to their relatively low cost and advanced homing capabilities [1]. Naval forces operating in cluttered littoral environments are particularly vulnerable from land or sea [1], [2]. Therefore, the susceptibility of Navy surface ships to asymmetric attack is strongly dependent on their capability to defend against the ASCM threat [2].

Wideband imaging radars, such as the inverse synthetic aperture radar (ISAR), are a unique class of ASCM targeting radar that provide range and bearing to a target as well as the general shape of the target [3], [4]. This enables an adversary to classify as well as detect a target. Consequently, an adversary using an imaging radar and a classification table can segregate high-priority targets from low priority targets or decoys [5]. Like other types of radars, imaging radars are susceptible to noise jamming [6]. However, noise jamming provokes an ISAR to use anti-jamming techniques or switch to alternative methods of target detection and classification [7]. In addition, future ASCMs are also expected to use imaging seekers in order to improve aimpoint accuracy and reject decoys [5].

Research on the next generation of techniques to counter imaging radars and seekers is in progress. A multi-chip electronic warfare (EW) system currently in development will have the capability to transmit synthesized signals to an imaging radar that, when integrated into the range-Doppler processing, form an image of a false target that will likely be indistinguishable from true target returns [8], [9]. Unlike noise jamming, the false-target EW system will not alert an interrogating ISAR to the ongoing deception. Consequently, the radar will continue to track both the real and false targets and will not

switch to alternative homing methods. An example of an imaging radar return and simulated false target generation on the USS Crockett is provided in Figure 1.1 (courtesy of the Tactical Electronic Warfare Division of the U.S. Naval Research Laboratory). The circuit that generates the false images in response to an imaging radar is called the Digital Image Synthesizer (DIS) [8].

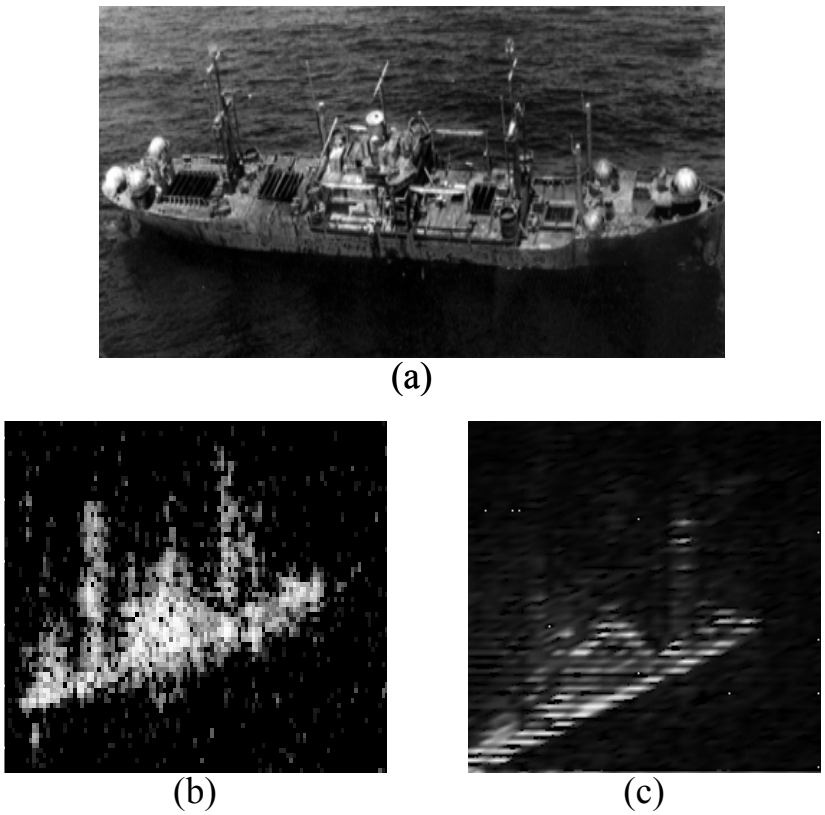

Figure 1.1 Imaging radar and false target generation example featuring (a) the USS Crocket, (b) AN/APS-137 imaging radar return of the USS Crockett, and (c) simulated output of an eight-bin digital image synthesizer circuit (After [9].).

B. PROBLEM STATEMENT

The design and development of effective and affordable electronic warfare systems is a difficult challenge that requires full exploitation of the most advanced technologies available. The future must include aggressive and innovative use of new technology

to produce systems that are operationally useful and supportable [10], [11]. Furthermore, the next generation of EW systems must be self-contained and capable of installation in miniature, low-power platforms such as expendable decoys and unmanned aerial vehicles (UAV) [12], [13]. In due course, the multi-chip EW system will give way to the EW system-on-a-chip (SoC) concept, where all analog and digital processing is performed in a single package. SoCs are now practical because of advances in mixed-signal design technology largely due to research in the cellular phone and wireless local area network (WLAN) fields [14]. For example, a sub-0.25 micron silicon germanium mixed-signal fabrication process has been used successfully for commercially available WLAN and cellular phone SoCs [14], [15].

This dissertation proposes a design for a high-performance, mixed-signal, EW SoC capable of producing false targets in response to wideband imaging radar interrogations. The minimum components required for the DIS EW SoC are an analog-to-digital converter (ADC), high-speed memory, the DIS signal processor, and a digital-to-analog converter (DAC). Each component in the SoC must be optimized for low-power and small die area while maintaining a high operating frequency. Unfortunately, SoC design involves the construction of an *entire* system and therefore it has many of the same problems as with any system design (e.g., independent module design, interface standardization, integration, etc.) [10], [11], [16]. Moreover, mixed-signal design is particularly difficult. There are few reliable mixed-signal system computer-aided design (CAD) tools such as a combined analog and digital hardware description language (HDL) [16]. Moreover, limited verification and simulation tools are available, especially mixed-signal circuit simulators capable of simulating analog circuits at the transistor level while simultaneously simulating digital circuits at the functional level in the time domain [10], [16]. Since it is extremely difficult and sometimes impossible to simulate the behavior of the entire SoC, each module must be tested independently and operate reliably regardless of alterations to other modules [11].

C. ORIGINAL CONTRIBUTION

The gargantuan task of design, simulation, and fabrication of an entire EW SoC is beyond the scope of this dissertation. Therefore, this research focuses on two key components of the SoC, the ADC and the DIS. The theory and circuit designs presented in this research leverage current advances in ADC design and mixed-signal fabrication technology to produce compact, power-efficient, high-performance EW SoC components.

New architectures and improvements are proposed for the DIS that resulted in a more compact, low-power design suitable for inclusion in an SoC design. A novel all-digital automatic clock-skew-control circuit design is introduced and correct operation verified with extensive simulation. A proof-of-concept chip implementing the skew control circuit was fabricated and tested to validate the hardware implementation of the design.

A high-speed, eight-bit robust symmetric number system (RSNS) folding ADC design is presented and was simulated in an advanced mixed-signal fabrication process. The ADC has lower power consumption and requires a smaller die area than comparable high-speed ADC designs. In support of the RSNS folding ADC, comprehensive mathematical and intuitive descriptions were developed for the robust symmetric number system.

The first RSNS-to-binary conversion algorithm is presented and converts the output of the RSNS ADC to a format compatible with most digital circuits. The RSNS-to-binary algorithm was designed and simulated in an advanced mixed-signal process. Simulation results verified the operation of the circuit and showed that the algorithmic approach to RSNS-to-binary conversion is orders of magnitude smaller and faster than conventional hardware conversion methods.

D. DISSERTATION OUTLINE

Chapter II describes and analyzes the existing DIS architecture. Several computational redundancies were discovered in the DIS architecture that, when removed, resulted in a DIS design with a significantly reduced layout area and reduced power consumption. In addition, it was shown that the clock distribution scheme in the current DIS requires manual skew control adjustment whenever a change is made to the circuit. Such a design procedure is not conducive to a mixed-signal SoC environment because of the difficulty of performing multiple full-chip simulations. A novel design was introduced for an automatic clock synchronization and skew control scheme that eliminated the necessity for repeated manual clock skew adjustment simulations. The result was decreased SoC design time and increased module reliability. Chapter II and its associated appendices provide the theory, circuit schematics, simulation results, and hardware testing for a proof-of-concept automatic clock synchronization chip.

Chapter III presents the circuit design for two robust symmetric number system (RSNS) folding ADCs. The RSNS is a symmetric residue number system that provides the theoretical basis for the design of high-speed folding ADCs with significant benefits in the areas of power, speed, and die area compared to other high-speed ADC designs. The mixed-signal fabrication process for the ADC circuit simulation is introduced and its benefits to SoC design are described. First, a three-channel six-bit RSNS folding ADC is presented. The ADC folding amplifiers, individual folding stages, and latched comparators are described in detail. The three-channel ADC design procedure was extended to construct a four-channel, eight-bit RSNS folding ADC that is suitable for converting the analog radar interrogation pulses to the eight-bit digital form required by the DIS. Comprehensive circuit schematics and simulation results are presented for both ADC designs.

Chapter IV extends the current two-modulus RSNS theory and analysis techniques. This chapter and its associated appendices provide the theory, closed-form analytic expressions, and search algorithm code to efficiently compute the size and location of the largest sequence of unique RSNS vectors for three-modulus and *N*-modulus systems.

Chapter V introduces two simple hardware schemes, a ROM and a decoder, to convert the digital thermometer code output of the RSNS folding ADC to binary format. The theory behind an alternative approach was developed that leads directly to an easily pipelined circuit occupying a die area that is orders of magnitude smaller than the other two approaches. The three-modulus RSNS-to-binary conversion process is presented first to illustrate the key conversion algorithm concepts. The three-modulus results are extended to generate an *N*-modulus RSNS-to-binary conversion procedure. Finally, the *N*-modulus conversion procedure was employed to design a four-modulus RSNS-to-binary circuit for the four-channel ADC in Chapter III. The four-modulus RSNS-to-binary converter provides the eight-bit binary output that meets the input requirements of the DIS circuit. Comprehensive circuit design schematics and verification simulation results are provided for both converter designs.

In Chapter VI, the results of the previous chapters are summarized and areas for future research are discussed.

II. DIGITAL IMAGE SYNTHESIZER ELECTRONIC WARFARE SYSTEM-ON-A-CHIP

A. SYSTEM-ON-A-CHIP ARCHITECTURE

The proposed design for a high-performance, mixed-signal, EW SoC capable of generating false targets in response to imaging radars interrogations is shown in Figure 2.1.

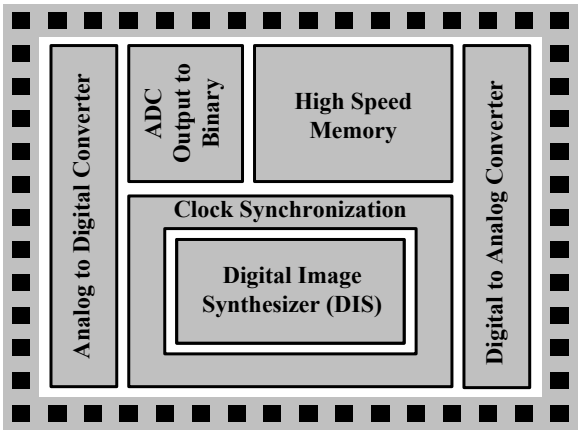

Figure 2.1 Components required for the DIS EW SoC.

The minimum components required to implement the DIS EW SoC are an analog-to-digital converter (ADC), high-speed memory, the DIS signal processor, a digital-to-analog converter (DAC), and a clock synchronization and distribution system. Each component in the SoC must be optimized for low-power and small die area while maintaining a high operating frequency. The remainder of this chapter proposes improvements to the DIS design that would make it more conducive to inclusion into the proposed SoC architecture.

B. DIGITAL IMAGE SYNTHESIZER DESIGN

The Digital Image Synthesizer (DIS) is the largest single component in the proposed EW single-chip architecture. The current DIS architecture contains at least two sources of redundant computation that, if the circuits are redesigned properly, could produce architectures with reduced layout area. Reducing layout area is critical since there will be several modules on the SoC and each one needs to be optimized for small die area and low power. The first section in this chapter presents some alternative DIS architectures with the goal of generating a design with a reduced die area.

Since the DIS is a pipelined synchronous machine, all of the range bins must be clocked in such a manner that the data arrives at each pipeline stage without corrupting the data from the previous pipeline stage. The second section in this chapter describes the architecture of the existing DIS and describes the current DIS resolution to this clocking problem, a technique known as counterflow-clock pipelining, and introduces a novel method for automatically controlling the clock skew inherent in the pipelining technique.

1. Current DIS Architecture

The DIS architecture is essentially a 512-tap finite impulse response (FIR) filter with each tap containing a complex modulation and is described in detail in [17]-[18]. The FIR signal processing circuit at each tap is called a *range bin*. The range bin gets its name from the fact that the time delay through the range bin processing elements represent an incremental target distance to an interrogating radar. Each range bin is composed of an adder, look-up table (LUT), gain multiplier, and two 16-bit adders. A conceptual diagram of the DIS showing only 2 of the 512 range bin processors (first and last) is presented in Figure 2.2. The dashed line indicates the extent of the existing single-chip DIS design.

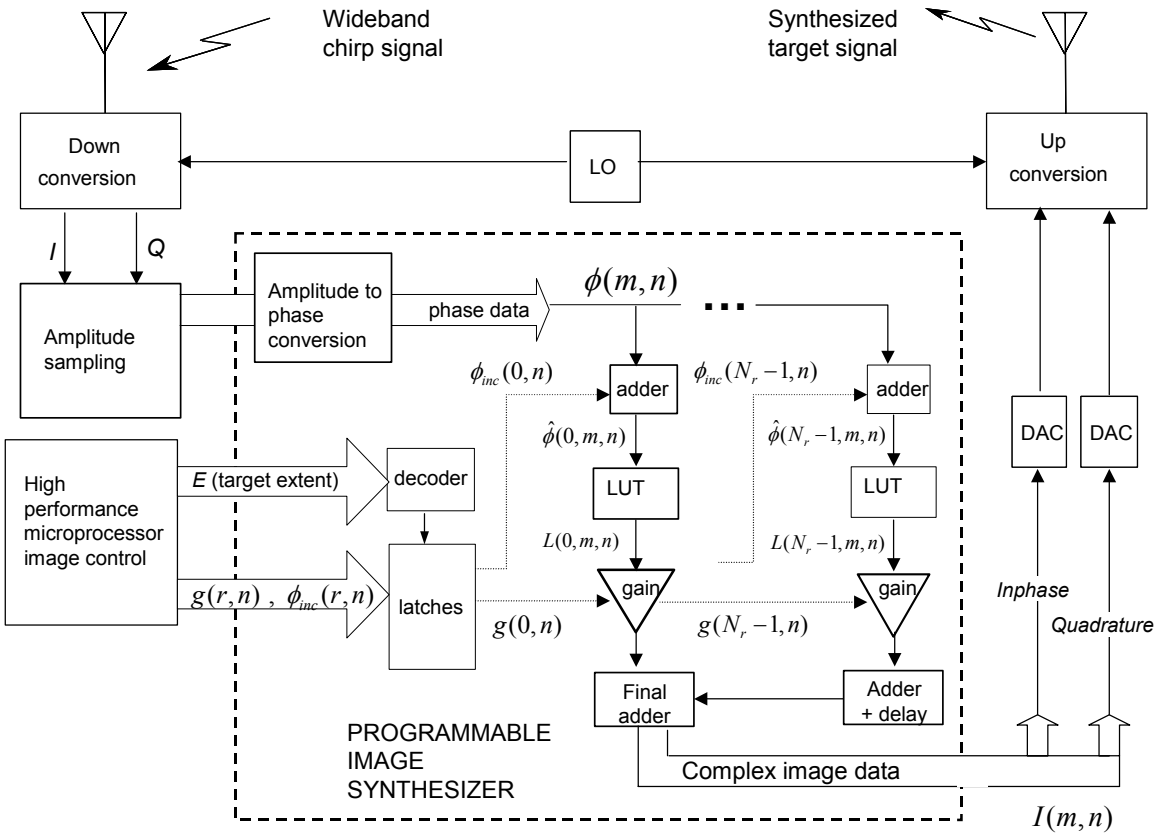

Figure 2.2 Single-chip DIS architecture (From [17].).

The integrated EW architecture proposed in this dissertation encompasses all of the components in the figure except for the up and down conversion circuitry, local oscillator, computer control, and antennas.

Since the DIS is the largest SoC component, it is the obvious focus of efforts to reduce die area in order to minimize the size and power requirements of the SoC. By far, the largest contributor to the size of the DIS is the size of the range bin circuit, which although relatively small, is replicated 512 times in the DIS circuit presented in [18]. Furthermore, increasing the number of range bins corresponds to an increase in the size of the false target generated by the DIS (for a constant clock rate), and would be necessary to generate larger false targets such as aircraft carriers. Figure 2.3 shows the detailed architecture of a single pipelined range bin. Figure 2.4 is a simplified representation of the DIS architecture showing 2 of the 512 range bins shaded in gray.

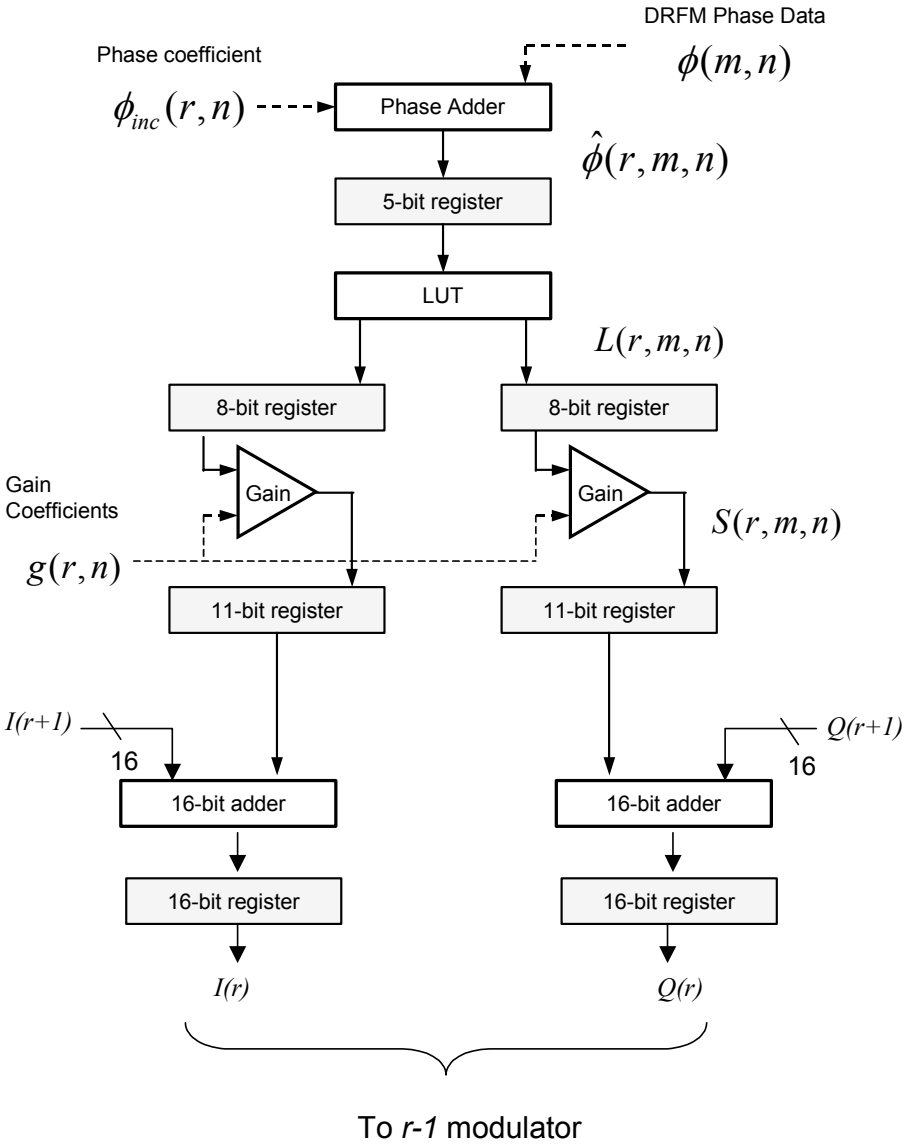

Figure 2.3 Architecture detail of single pipelined range bin (From [18].).

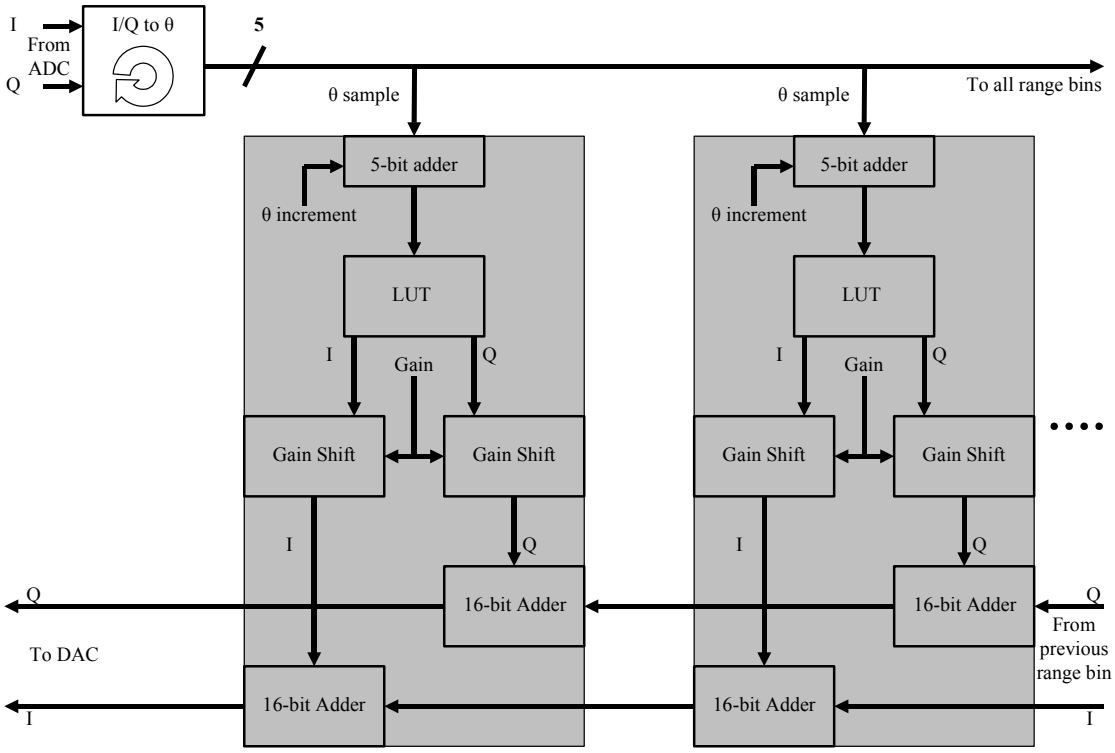

Figure 2.4 Simplified DIS architecture.

The inphase and quadrature (*I/Q)* radar pulse samples from the ADC in Figure 2.2 are converted to a phase angle sample and then distributed in parallel to each range bin. The *I/Q*-to-phase-angle converter shown in Figure 2.4 and Figure 2.2 is an implementation of an unrolled CORDIC (Coordinate Rotation Digital Computer) algorithm described in detail in [17]. Within each range bin, on every clock pulse a five-bit phase increment is added to the phase angle samples (Doppler processing) and the result is converted back to *I/Q* using an LUT. The *I/Q* signals from the LUT are multiplied by a gain factor (radar cross section processing), then summed with the result from the adjacent range bin (which includes tapped delay) and the result is passed forward. The five-bit phase angle increment and gain factor are not necessarily the same from range bin to range bin and depend on the image being synthesized. The following sections address the computational redundancies in the DIS architecture and presents alternative architectures that remove the redundancies.

2. Alternative DIS Architectures

The first computational redundancy addressed in this section is the conversion from *I/Q* to phase angle (for Doppler processing) and the subsequent conversion from phase angle back to *I/Q*. From a logical perspective, it seemed probable that eliminating the phase angle conversion and performing all computations with the *I/Q* samples would simplify the circuit and reduce layout area. The second redundancy addressed stems from the fact that each of the 512 range bins increments the phase angle by a 5-bit value. Thus, since a 5-bit binary number has only 32 unique values, at least 480 of the 512 range bins are performing redundant calculations on each clock cycle.

Figure 2.5 shows an alternative architecture that eliminates the phase angle conversion. In this architecture, the phase adder and LUT in each range bin were replaced by a mathematically equivalent *I/Q* rotation operation. The process of using the CORDIC algorithm to convert *I/Q* to five-bit phase angle requires almost exactly the same hardware as rotating the *I/Q* vectors by a five-bit phase angle [19]. The only significant difference in the algorithm is the initialization values. Therefore, the transistor count for the *I/Q* to phase angle converter from [17] will be used as an approximation for the transistor count of the I/Q vector rotator circuit. Unfortunately, the *I/Q* rotation circuit as presented in [17] requires almost 50,000 transistors, which is 33 times larger than the phase adder and LUT. Therefore, although the architecture in Figure 2.5 appears less complex than the original DIS architecture, it has approximately 33 times more transistors and no apparent increase in speed.

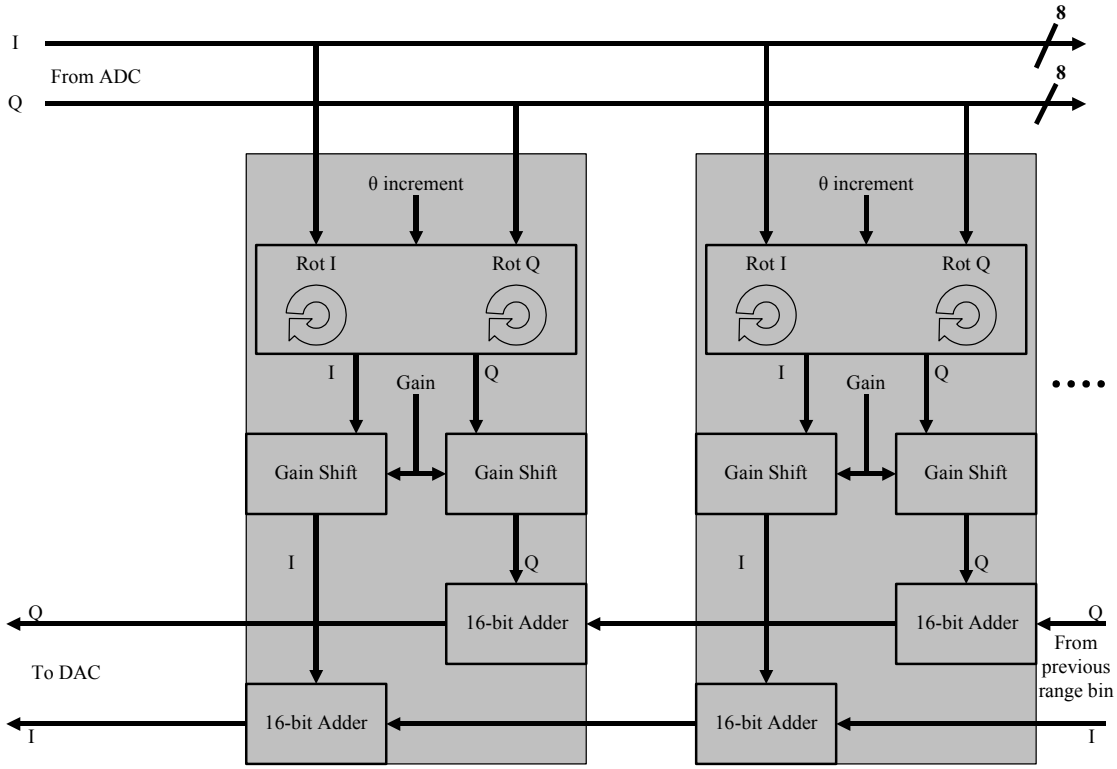

Figure 2.5 DIS architecture using range bin I/Q rotation.

Figure 2.6 represents an alternative DIS architecture that eliminates both phase angle conversion and redundant range bin phase increment addition. In this design, each *I/Q* sample pair is fed into 31 parallel *I/Q* rotation circuits, which simultaneously compute all possible 31 rotations of the input sample. The rotated samples are then distributed to the 512 range bins via a distribution network (bus, tree, butterfly, etc.). The range bins shown in gray are a mere half of their original size in terms of transistor count. However, computing the savings in terms of transistors for the entire DIS circuit yields a negative result. The 31 *I/Q* rotation circuits add 1,550,000 transistors, while the elimination of 512 phase adders and LUTs remove only 768,000 transistors. This computation ignores the overhead in wiring and switches necessary for the distribution network, which will not be negligible. Therefore, this architecture does not provide any savings in terms of die area until the number of range bins reaches 1024 or more. Even if there were more than 1024 range bins, the distribution network would most likely require so much additional wiring and switching circuitry as to make this architecture impractical.

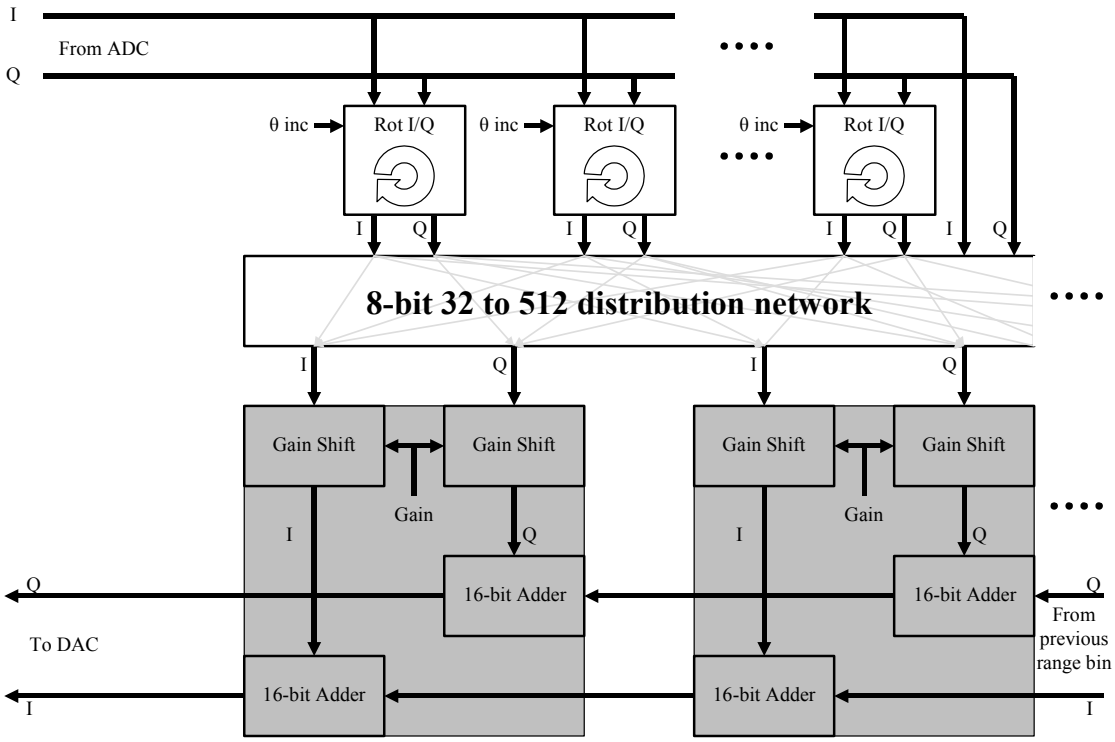

Figure 2.6 DIS architecture with 31 I/Q rotations and distribution network.

An interesting observation about this architecture is that the *I* and *Q* portions of the range bins are decoupled. That is, the range bins can be split into *I* range bins and *Q* range bins that can be physically separated into independent pipelines. Therefore, this architecture may be useful if layout flexibility is important.

Figure 2.7 represents a final alternative DIS architecture that eliminates both phase angle conversion and redundant range bin phase increment addition. In this design, each *I/Q* sample pair is fed into an *I/Q* rotation circuit. The rotation circuit converts the *I/Q* samples to a phase angle sample as in the original DIS architecture. The phase angle sample is then passed to an incrementer, which increases the phase angle by one on each clock cycle. The incremented phase angle is then converted back to *I/Q* with a LUT and the result is placed onto a high-speed *I/Q increment* bus. A second incrementer is used to keep track of the number of increments applied to the phase angle sample. The count of the second incrementer is placed on a second high-speed bus. Both busses connect to every range bin in parallel. The latch in each range bin captures the value on the *I/Q* bus

when the pre-loaded latch code in the particular range bin matches the increment count on the second bus.

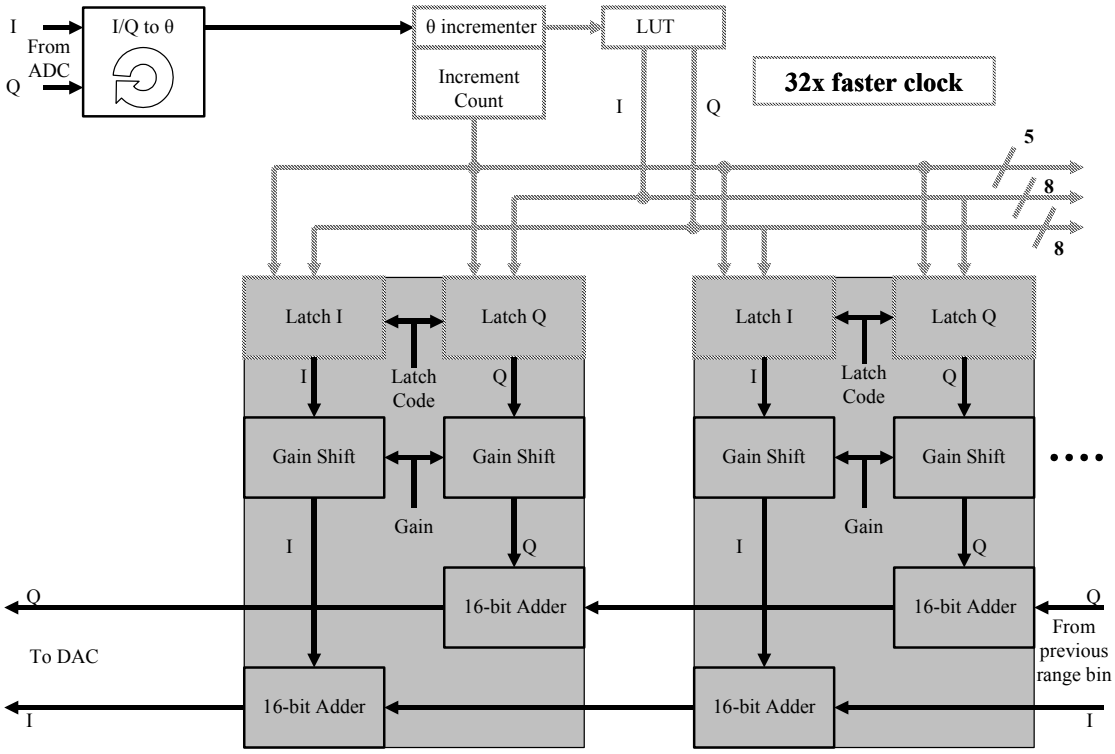

Figure 2.7 DIS architecture with a phase angle incrementer and bus distribution.

The transistor count of each range bin is reduced by approximately one-third and, like the previous architecture, the *I* and *Q* portions of the range bin are decoupled providing more layout flexibility. The downside of this architecture is that the portion of the design drawn with patterned lines (i.e., incrementers, LUT, distribution buses, and range bin latches) must be clocked 32 times faster than the rest of the circuit. This must be done to maintain the same overall clock rate as the original DIS architecture.

A five-bit version of the fast incrementer in [20] was designed and simulated in the same fabrication process as the DIS to test the practicality of the architecture in Figure 2.7. A schematic of the high-speed incrementer is provided in Figure 2.8.

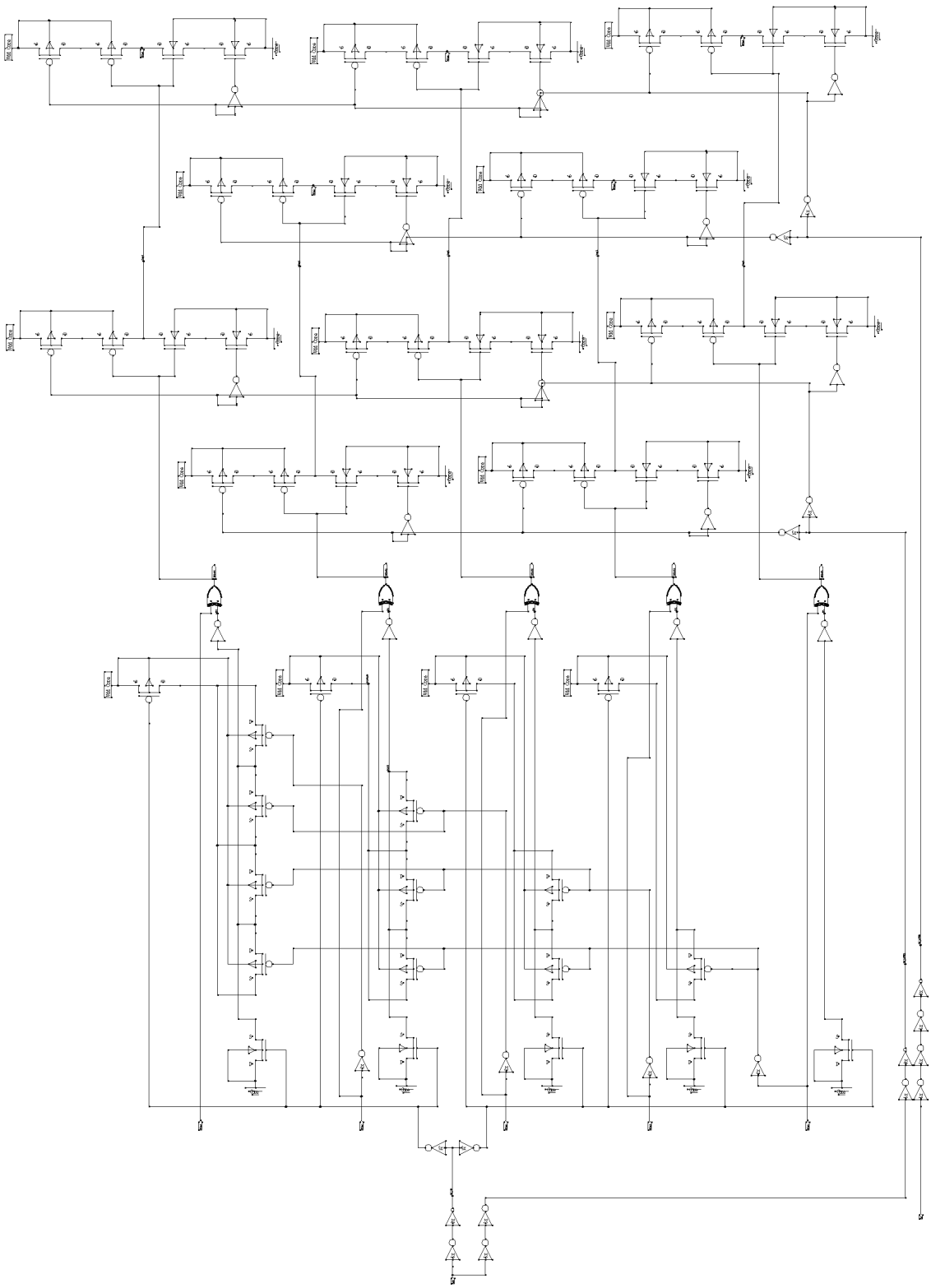

Figure 2.8 High-speed incrementer circuit schematic.

The lower half of the schematic shows the priority-based incrementer and the upper half of the schematic shows two levels of dynamic latches that feed the incrementer output back to the input. The circuit was designed and simulated using Tanner Research software. The simulation parameters were provided by MOSIS (Metal-Oxide Semiconductor Implementation System). MOSIS is a low-cost prototyping and small-volume production service for VLSI (Very Large Scale Integration) circuit development at the University of Southern California with support from the National Science Foundation. MOSIS keeps the cost of fabricating prototype quantities low by gathering together multiple projects into one fabrication run. This allows customers to share overhead costs associated with mask making, wafer fabrication, and assembly. The fabrication process used for simulation was the Taiwan Semiconductor TSMC CL018 process used to simulate the existing DIS schematic. The TSMC process is for 1.8-volt applications and has a thick oxide layer for making 3.3-volt transistors and is described in detail in [17]. Simulation results for the circuit are provided in Figure 2.9.

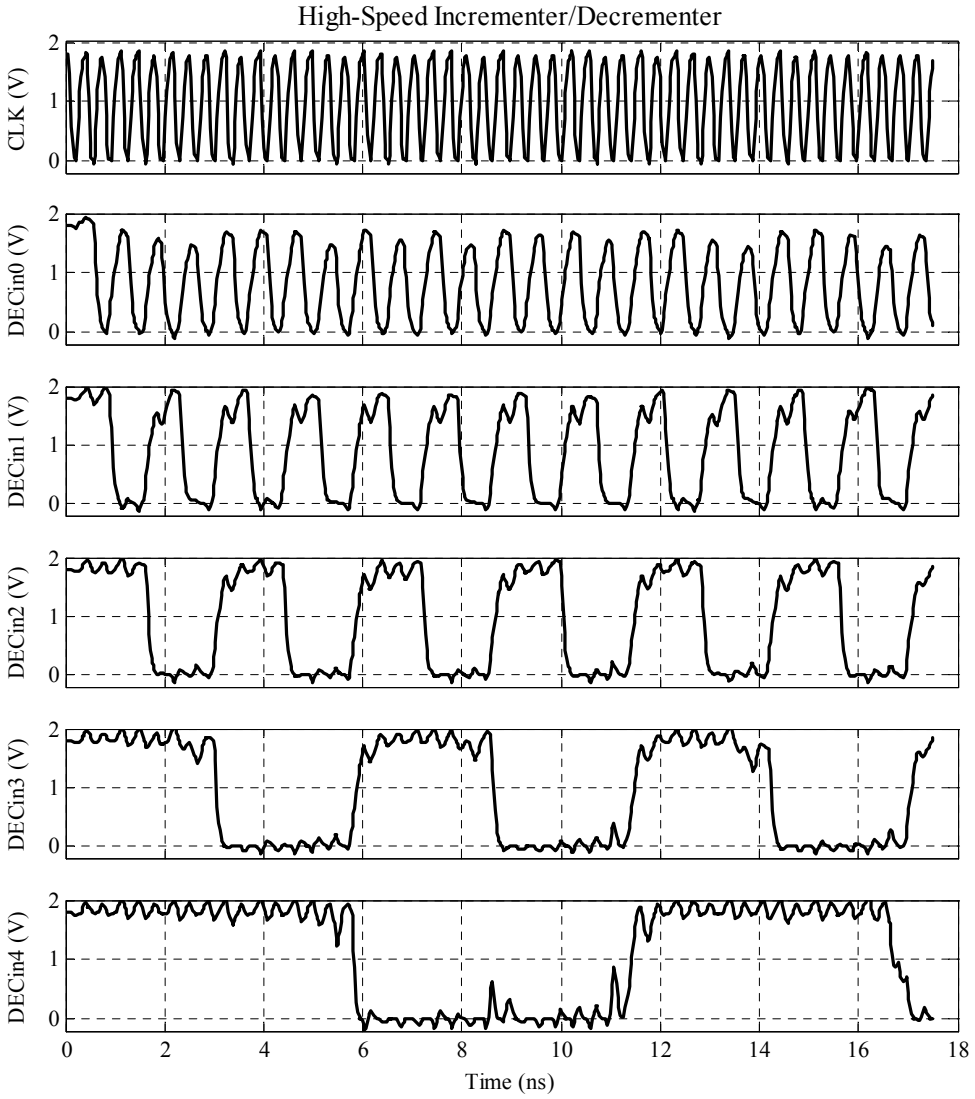

Figure 2.9 Incrementer simulation results.

Correct operation of the circuit was achieved up to a frequency of 2.85 GHz. The DIS must operate at a minimum frequency of 500 MHz in order to be effective against modern imaging radars [18]. Thus, the fast incrementer must operate at a frequency of at least $500 \times 32 = 16,000$ MHz, or 16 GHz. Thus, the incrementer is approximately six times slower than necessary. In order for this architecture to be practical against modern imaging radars, the design must be implemented in a fabrication process that allows for the incrementer to run at a pipelined speed greater than 16 GHz.

In summary, although the alternative DIS architectures presented in this section do not represent all the possible improvements to the original DIS architecture, this research demonstrates that it is unlikely that removing the redundant range bin phase addition reduces the transistor count in the DIS circuit. The redundant addition of the phase increment in each range bin can be eliminated by pre-computing 31 phase increments. However, the problem of distributing the 32 samples to each range bin in a single DIS clock cycle requires either a large and complex distribution system or a high-speed bus. However, both distribution methods were proven impractical in the current DIS fabrication process. Furthermore, revising the DIS circuit to eliminate the phase conversion did not provide the anticipated layout area reduction. The sheer size of the *I/Q* rotator circuit prevents its use in the range bin and, as long as the number of range bins remains relatively low, the phase angle conversion, although redundant, produces a circuit with a minimal transistor count.

While the transistor count of the DIS was not reduced using the architecture designs presented in this section, the DIS has a pipeline clock skew control scheme that is not conducive to the mixed-signal SoC environment. The DIS architecture improvement in the next section provides a novel and effective solution for automatically controlling the DIS pipeline clock skew.

C. DIS CLOCK-SKEW CONTROL

1. Counterflow-Clock Pipelining

Since the DIS is a pipelined synchronous machine, all of the pipeline latches must be clocked in a manner such that the data in one stage does not corrupt the data in the following pipeline stage. The obvious solution is to clock each pipeline latch at exactly the same instant at an appropriate frequency. From a practical sense, this is an impossible task and realistically each pipeline latch is clocked with some skew relative to the other latches. There are many methods for reducing clock skew [21] but one of the most prevalent is the H-tree method shown in Figure 2.10.

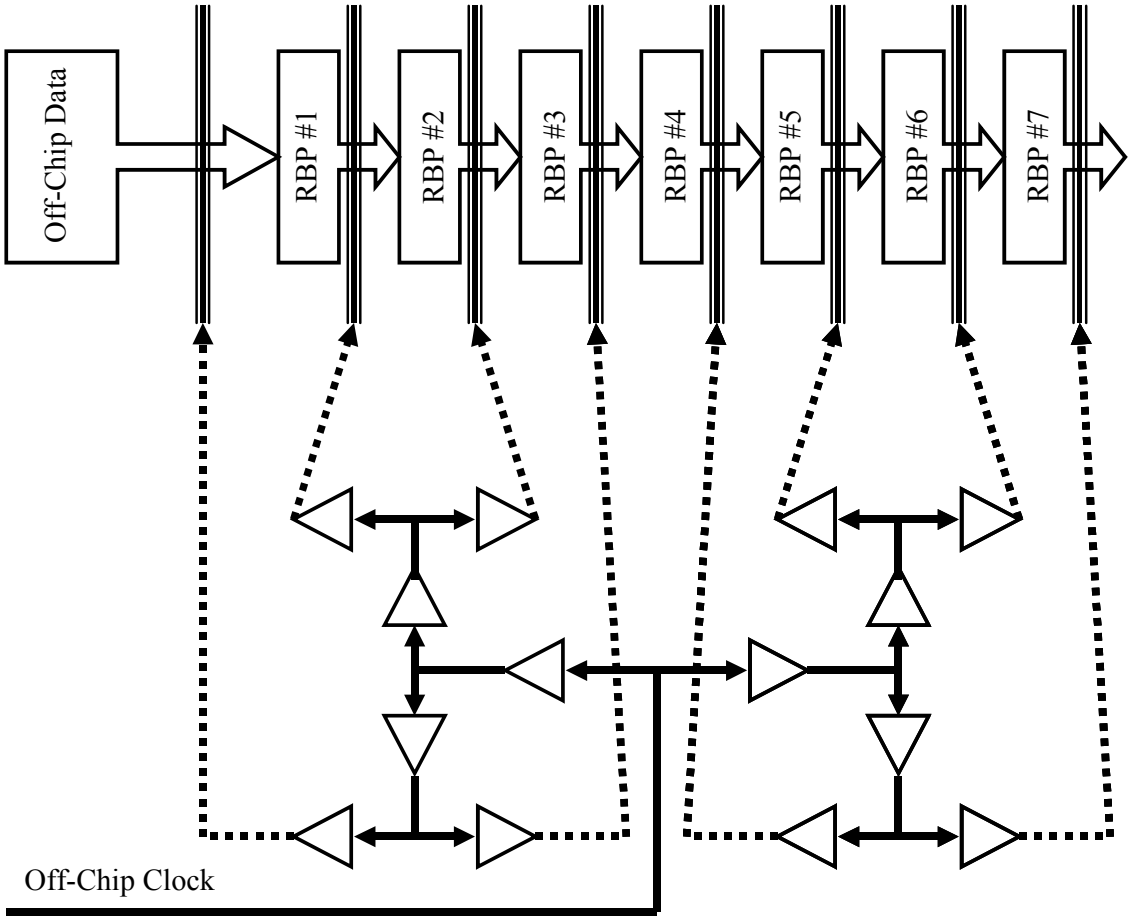

Figure 2.10 H-tree clock distribution method.

The clock delay path to each range bin pipeline latch, represented by the patterned lines, is theoretically identical. In practice however, the clock distribution wiring is rarely exactly the same length and process variations across the chip impart variable delay on the buffers. Although effective at limiting skew, the H-tree has the side effect of clocking all latches at the same time causing ground bounce and power supply drop, which can be disastrous to chips with low noise margins [21]. An alternative clocking scheme particularly effective on architectures such as the DIS with primarily unidirectional data flow is a technique known as counterflow-clock (C^2) pipelining [21]. The advantages of C^2 pipelining include application of simple local timing constraints independent of clock skew and reduced switching noise due to fewer simultaneously clocked switches. Although there are several ways to implement C^2 pipelining, the current DIS architecture implements the C^2 pipelining scheme in a manner depicted in Figure 2.11.

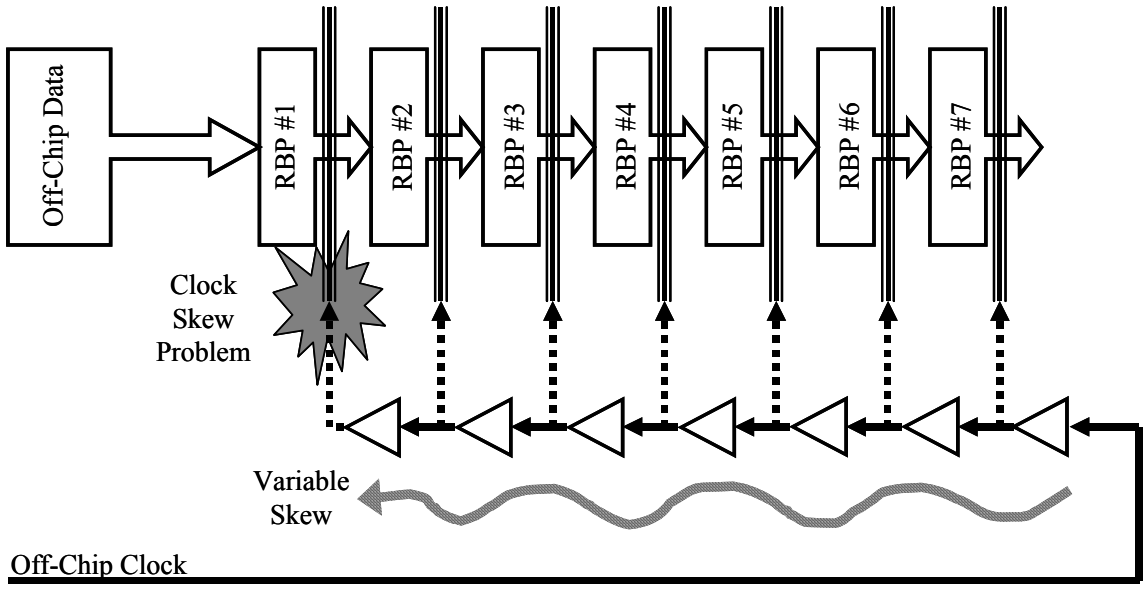

Figure 2.11 Counterflow-clock distribution method without skew control.

Notice that the clock signal flows in a direction opposite to the data. The only clock skew problem occurs at the point where the off-chip data, driven by an off-chip clock, coincides with the counterflow clock. Depending on the number of range bins, wiring delay, and number of buffers, the two clocks could be out of phase by as much as 180 degrees. The way the current DIS architecture solved the skew problem is by using manually adjusted variable delay elements in each range bin as shown in Figure 2.12.

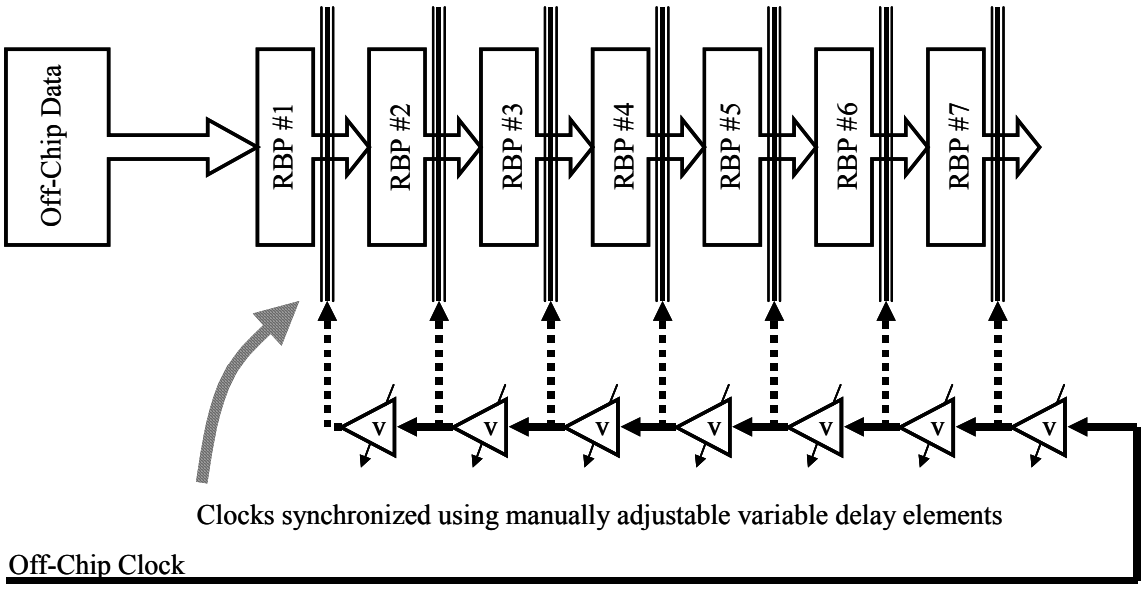

Figure 2.12 Counterflow-clock distribution method with skew control.

Synchronization is achieved by measuring clock skew with an oscilloscope and adjusting the variable clock delay in each range bin until the counterflow clock is synchronized with the off-chip clock at the coincidence point. The problem with this method is that it requires time consuming human involvement to perform the measurement and adjustment. Also, the skew adjustment is static; thus if circuit timing drifts due to temperature, voltage changes, or component aging, the manual adjustment does not automatically compensate. The variable delay element in each range bin is shown in Figure 2.13. The clock delay is adjustable by an amount equal to the delay through two minimum-sized inverters.

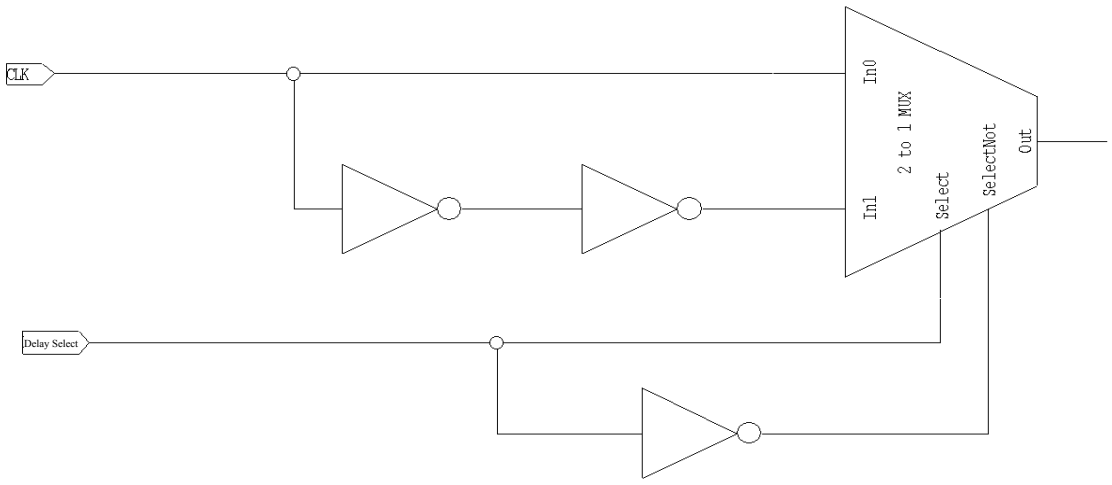

Figure 2.13 Manual skew control method in original DIS architecture.

Much research has been published on clock synchronization using static [22] and dynamic [23]-[25] techniques. All of the dynamic methods published to date have employed the use of analog circuits as either synchronization detection or correction mechanisms. The next section in this chapter introduces a novel all-digital method for automatically and dynamically synchronizing the off-chip and counterflow clock signals. The method also reduces the clock skew caused by variations in chip parameters.

2. Automatic Synchronization Approach

For the purposes of this research, clock skew is defined as the absolute value of the time difference between the positive rising edges of two clock signals. One clock signal is considered the reference clock and the skew of a second clock signal is measured relative to the reference clock. This concept is illustrated in Figure 2.14.

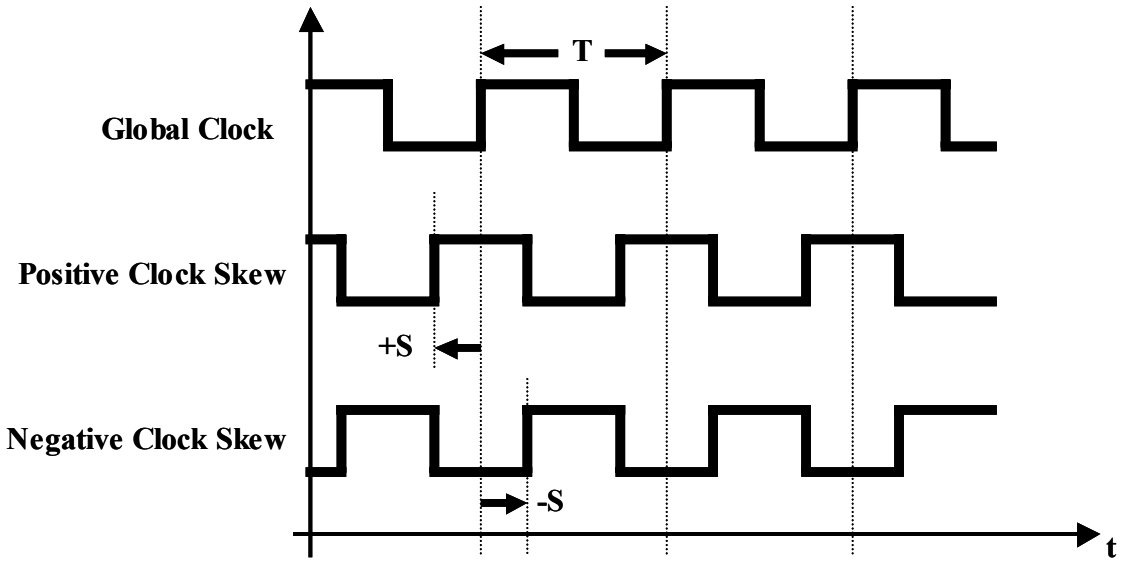

Figure 2.14 Clock skew diagram.

In the diagram, the reference clock is the *global clock*, referring to the concept that the *global clock* is the reference for other synchronous chips or modules. Relative to the *global clock*, the remaining two clock signals show positive clock skew, where the leading edge of the skewed clock occurs before the positive edge of the *global clock*, and negative clock skew, where the positive edge of the skewed clock occurs after the positive edge of the *global clock*. Consequently, the skew limits are

$$0 > |S| \geq \frac{T}{2}, \tag{2.1}$$

where T is the clock period and S is clock skew. Although not possible in a practical sense, clock signals with a relative skew of exactly $T/2$ can be defined as either positive or negative depending on the application. The clock signals presented in Figure 2.14 have a 50% duty cycle; however the clock signals considered in this research are not limited to a 50% duty cycle nor must they both have the same duty cycle for the purposes of the synchronization method presented in this dissertation.

The clock synchronization scheme presented in this research consists of three components: a *phase-check module*, a *phase-adjustment module*, and a *control module*. This scheme is shown in Figure 2.15.

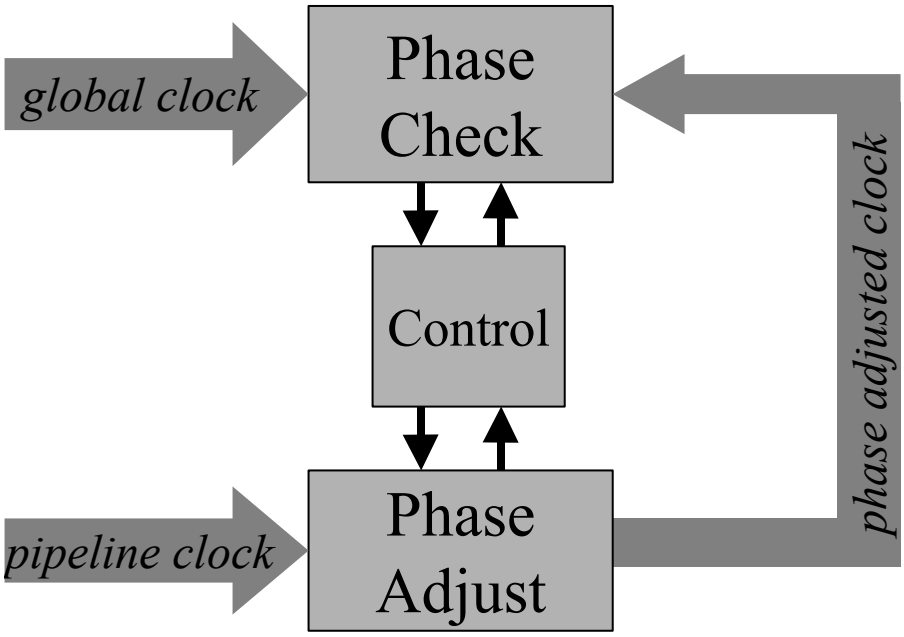

Figure 2.15 Clock phase synchronization diagram.

The phase-check module compares two clock signals and determines if they are in phase within a certain tolerance. If the two clock signals are out of phase, the phase-check module also determines whether the C^2 pipeline clock has positive or negative skew relative to the *global clock*. If the phase-check module determines that the *global clock* and C^2 pipeline clock are out of phase, the phase-adjustment module incrementally increases or decreases the delay applied to the C^2 pipeline clock based on the sign of the phase difference. The control module regulates the timing of the phase-check/phase-adjustment cycle such that the full effect of the incremental phase adjustment influences the C^2 pipeline clock before the next phase check occurs. Since the phase-adjustment module adjusts the phase of the skewed clock in increments, several phase-check/phase-adjustment cycles may be required to synchronize the two clocks. For notational purposes, the adjustment increment is defined as 2τ seconds, where τ is the average delay of a minimum-sized inverter. To avoid infinite adjustment oscillations, the phase-check module must detect a skew with an absolute value less than or equal to τ as *in-phase* and a skew with an absolute value greater than τ as *out of phase*.

3. Circuit Design

The block diagram in Figure 2.16 presents a more detailed design of the clock synchronization scheme described in the previous section. In this implementation, the phase-adjustment module is composed of a counter driving a variable delay module. The variable delay module represents the variable delay elements contained in the range bins shown in Figure 2.13. The control module is implemented with a finite state machine (FSM). The phase-check module design is the same as described in the previous section. Starting with the *global clock* and the *pipeline clock* signals entering from the left of the figure, the data flow is through the blocks in a clockwise fashion.

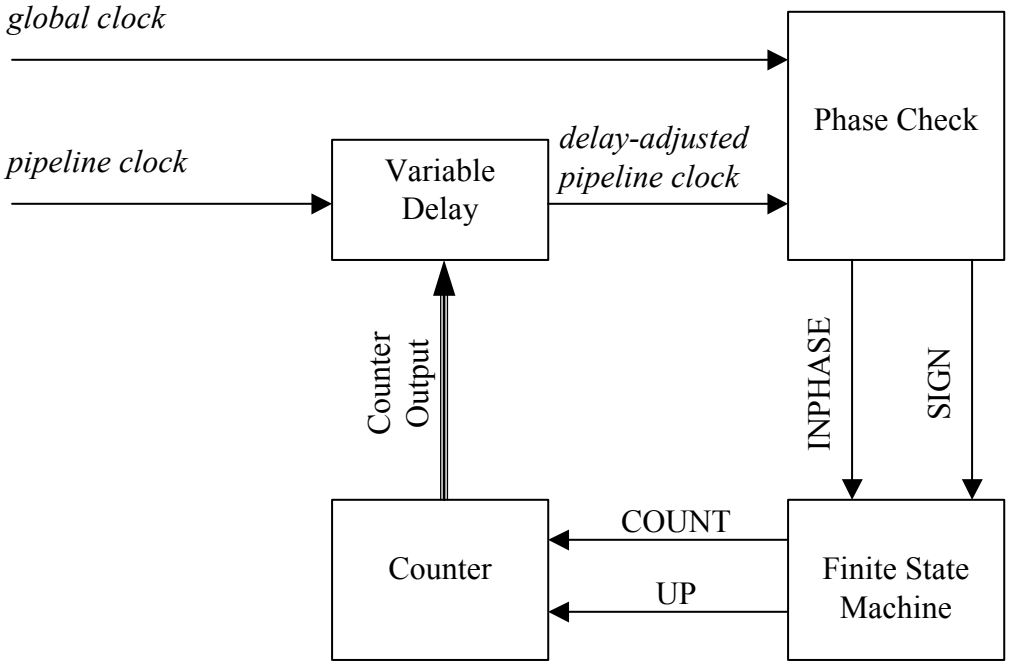

Figure 2.16 Block diagram of a simple clock synchronization scheme.

Stepping through the data flow diagram, the *global clock* enters the phase-check module directly while the *pipeline clock* first receives some delay in the variable delay module. The *global clock* and delay-adjusted pipeline clock (DPCLK) are inputs to the phase-check module. The phase-check module determines whether the *global clock* and

the *delay-adjusted pipeline clock* are synchronous within a certain tolerance, τ. The output of the phase-check module is an INPHASE signal, which is asserted only if the two clock signals are synchronous, and a SIGN signal, which is asserted if the *delay-adjusted pipeline clock* skew is positive with respect to the *global clock* and negated if the *delay-adjusted pipeline clock* skew is negative with respect to the *global clock*. The FSM provides the correct timing for the clock phase synchronization cycle as well as COUNT and UP signals to the counter. These two signals enable the counter to count up, count down, or hold the current count. The binary value of the counter determines the amount of delay added to the *pipeline clock* in the variable delay module.

A more comprehensive block diagram of the clock synchronization circuit using generic clock labels is provided in Figure 2.17. The second variable delay circuit is added for testing purposes to impart controlled skew on the Clock B signal. The operation and schematics of each module are presented in detail in the following sections.

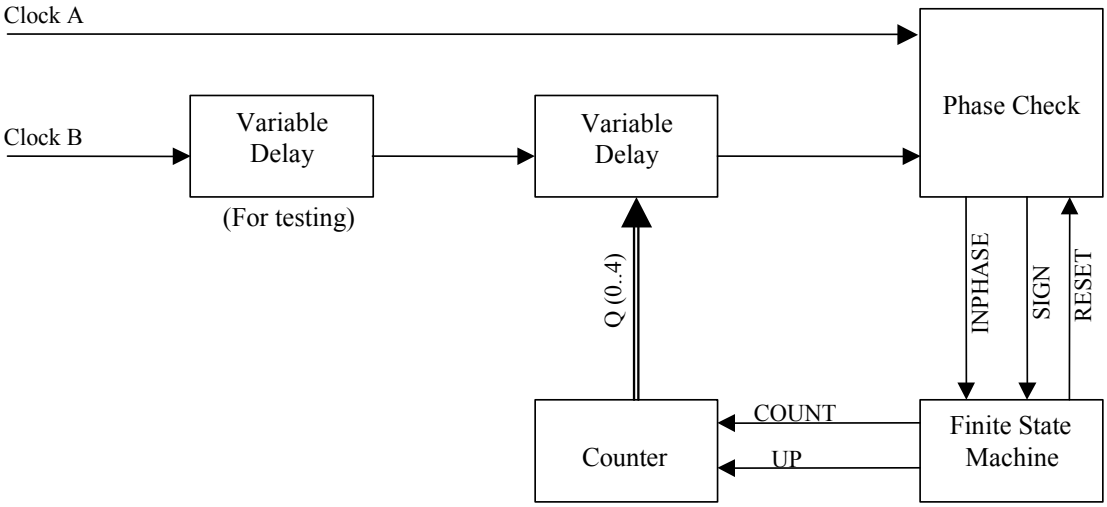

Figure 2.17 Clock synchronization design diagram.

a. Phase-Check Module

The heart of the clock synchronization scheme is the phase-check module. The module has four inputs as shown in the schematic in Figure 2.18.

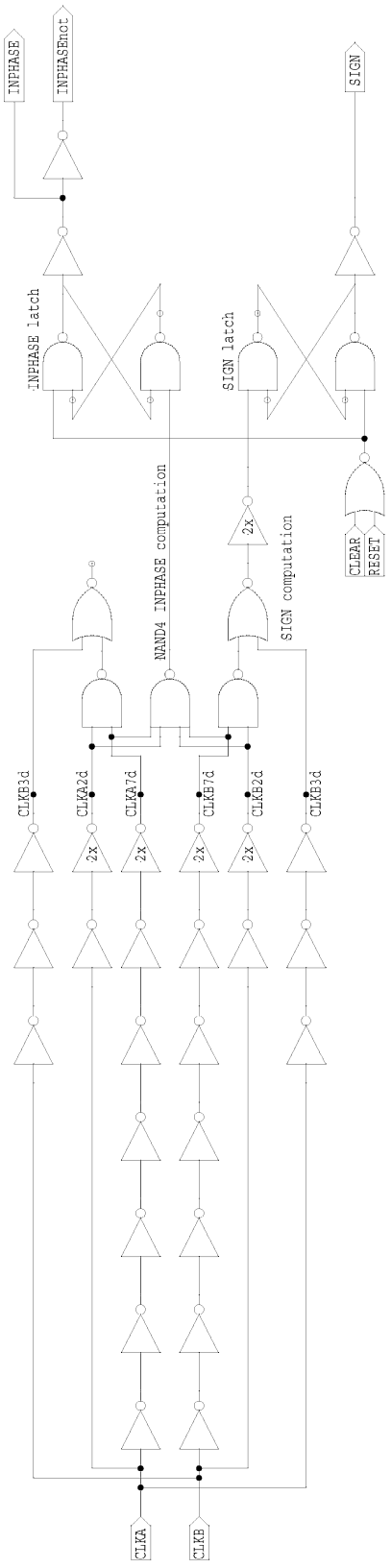

Figure 2.18 Phase-check module schematic.

All circuit schematics presented in this section were constructed using Tanner Research software described in [17]. In Figure 2.18, the first two inputs are clock signals labeled CLKA and CLKB, which represent the *global clock* and C^2 pipeline clock respectively. The third input is a RESET signal from the FSM, and the last input is a chip-wide CLEAR signal that, when asserted, returns all modules to a known initial state. The two outputs of the phase-check module are the INPHASE and SIGN signals described in the previous section. The four-input NAND gate (NAND4) in the center of the schematic determines whether the two clock signals are synchronized. The top two inputs to the NAND4 are the CLKA signal with a two-inverter delay (CLKA2t) and an inverted CLKA signal with a seven-inverter delay (CLKA7t). The bottom two inputs to the NAND4 are the same as top two inputs using the CLKB signal. The three inverter gates and the NOR gate with no output connection at the top of the schematic are included for electrical load balancing. Each pair of inputs has the timing shown in Figure 2.19.

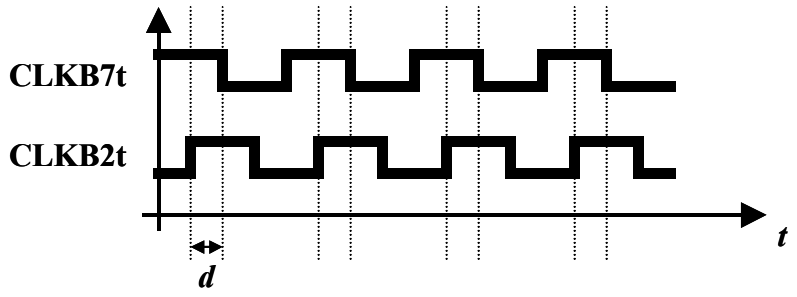

Figure 2.19 Timing diagram for one pair of inputs to the NAND4 gate.

The time between the rising edge of the delayed clock and the falling edge of the inverted clock is a constant, *d*. The parameter *d* also corresponds to the amount of time that the pair of signals are both in the logic *high* state. Defining a single inverter delay as τ, the value of *d* in this case is 5τ. When the clocks are perfectly synchronous, combining all four inputs with the NAND4 yields the timing diagram shown in Figure 2.20.

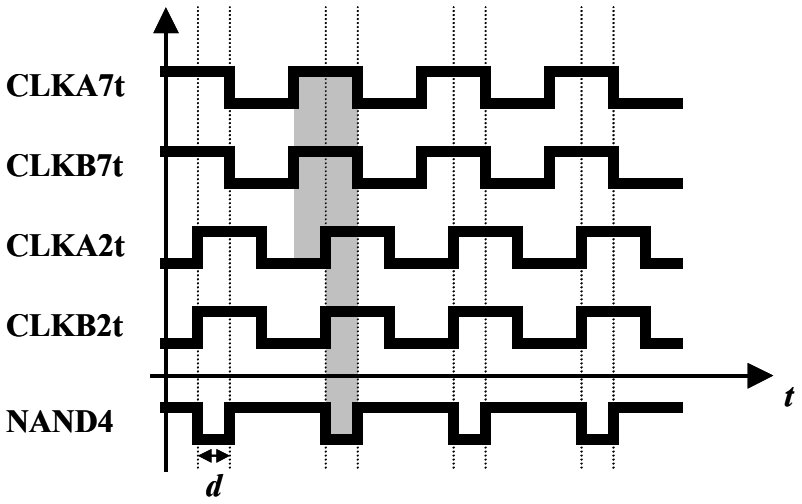

Figure 2.20 Timing diagram for the NAND4 output for synchronous clocks.

The shaded areas in the diagram show which edge of each input affects the NAND4 output. Notice that in the NAND4 output, a downward pulse of width *d* indicates that the clocks are synchronous. Figure 2.21 shows the NAND4 output for the case where CLKB has some negative skew *S*, which is less than *d*.

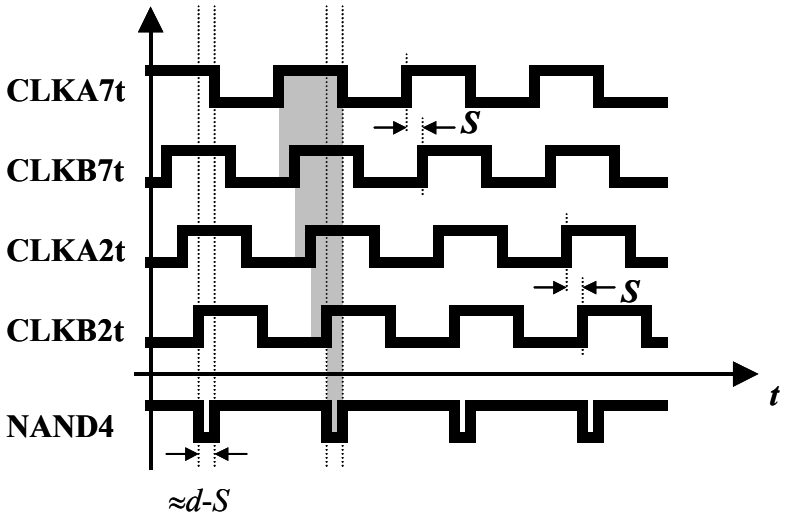

Figure 2.21 NAND4 output for clocks with skew $S < d$.

The NAND4 output still shows a synchronization pulse, but the pulse width is $d - S$. Observe that when skew is applied to CLKB, both the inverted and delayed versions of CLKB are displaced by the same skew amount. The negative-going leading edge of the NAND4 output synchronization pulse is determined by the second

leading edge of the two-inverter delayed clocks (A or B), while the positive-going trailing edge of the synchronization pulse is determined by the first falling edge of the seven-inverter delayed clocks (A or B). Therefore, if the skew *S* becomes greater than *d*, there is no synchronization pulse as shown in Figure 2.22. The latched output of the NAND4 is the INPHASE signal and is one of the two phase-check module outputs.

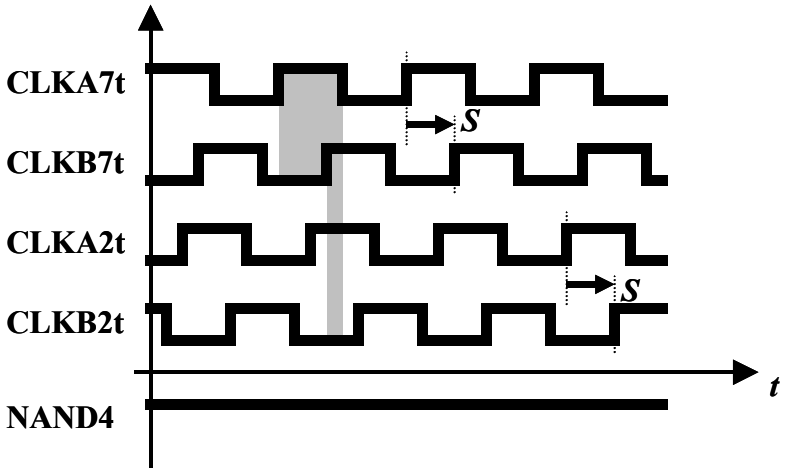

Figure 2.22 NAND4 output for clocks with skew $S > d$.

The other phase-check module output is the SIGN signal that represents the sign of the clock skew. The SIGN signal is computed by the two-input NAND and two-input OR gates in the center of Figure 2.18. If the two clock signals are in phase, then the sign of the skew is meaningless. However, if the two clock signals are not in phase, the sign of the skew is computed by combining the delayed CLKB signals in Figure 2.19 with a three-inverter delayed CLKA signal, CLKA3t. The logic *high* states of CLKB2t and CLKB7t always overlap by *d*. Thus, the two-input NAND gate produces a pulse of width *d* when CLKB2t and CLKB7t are the inputs. The output of the two-input NAND gate is used as one input to a two-input OR gate with CLKA3t as the other input. When the skew is negative, the output of the OR gate is a pulse as shown in Figure 2.23.

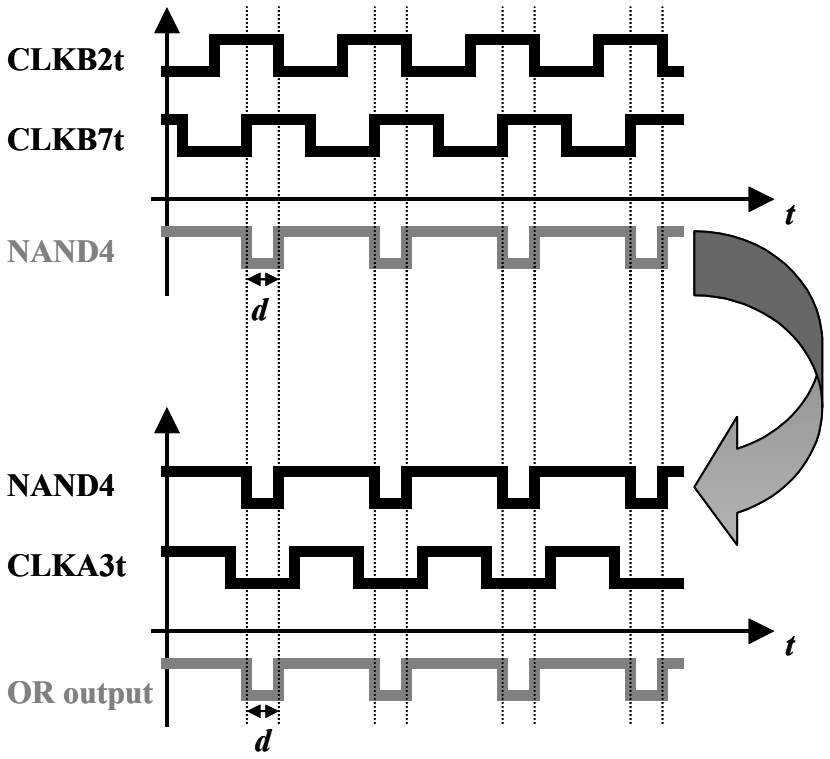

Figure 2.23 Timing diagram showing clock skew sign computation for negative skew.

When the skew is positive, the output of the OR gates is a constant logic level as shown in Figure 2.24. The latched output of the OR gate is the SIGN signal and is the second phase-check module output.

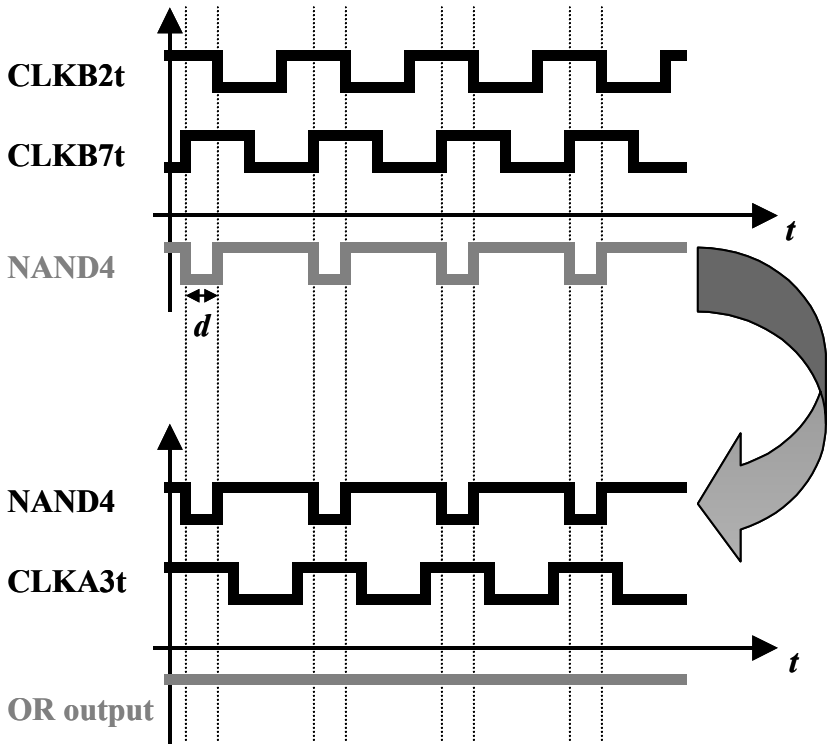

Figure 2.24 Timing diagram showing clock skew sign computation for positive skew.

The sign of the clock skew changes at only two points. One is when $S = 0$ and the other is when $S = T/2$, where T is the clock period. At these two points, the exact sign of the skew is ambiguous. The sign ambiguity at $S = 0$ is inconsequential because the two clocks are synchronized and the sign of the skew is meaningless. However, the ambiguity at $S = T/2$ is still a concern and will be solved by an initialization technique described in following sections.

It was determined in a previous section that, in order to avoid infinite phase adjustment oscillations, the phase-check module must detect a skew with an absolute value less than or equal to τ as *in-phase* and a skew with an absolute value greater than τ as *out of phase*. In order for the phase-check circuit to realize these tolerances, the transistors composing the NAND4 gate must be sized appropriately, or *tuned*. The size of the transistors affects the NAND4 gate sensitivity and determines the synchronization threshold. Therefore, the clock-synchronization circuit is not process independent. However, the phase-check module is a very small circuit that was easy to simulate. The simulation results helped determine the correct transistor sizing for the NAND4 as well

as the optimum number of delay inverters preceding the NAND4 for the desired operating frequency. The phase-check tuning simulation is much shorter than the simulation for the entire DIS circuit used to manually tune the variable delay elements. Furthermore, tuning the phase-check module needs to be performed only once, regardless of any subsequent changes to the DIS architecture.

b. Finite State Machine

The FSM provides the timing for the COUNT and UP signals used by the counter module as well as the RESET signal used by the phase-check circuit. The timing is set such the phase adjustment is manifest on the pipeline clock before the next phase-check cycle. Figure 2.25 provides the state transition diagram for the FSM.

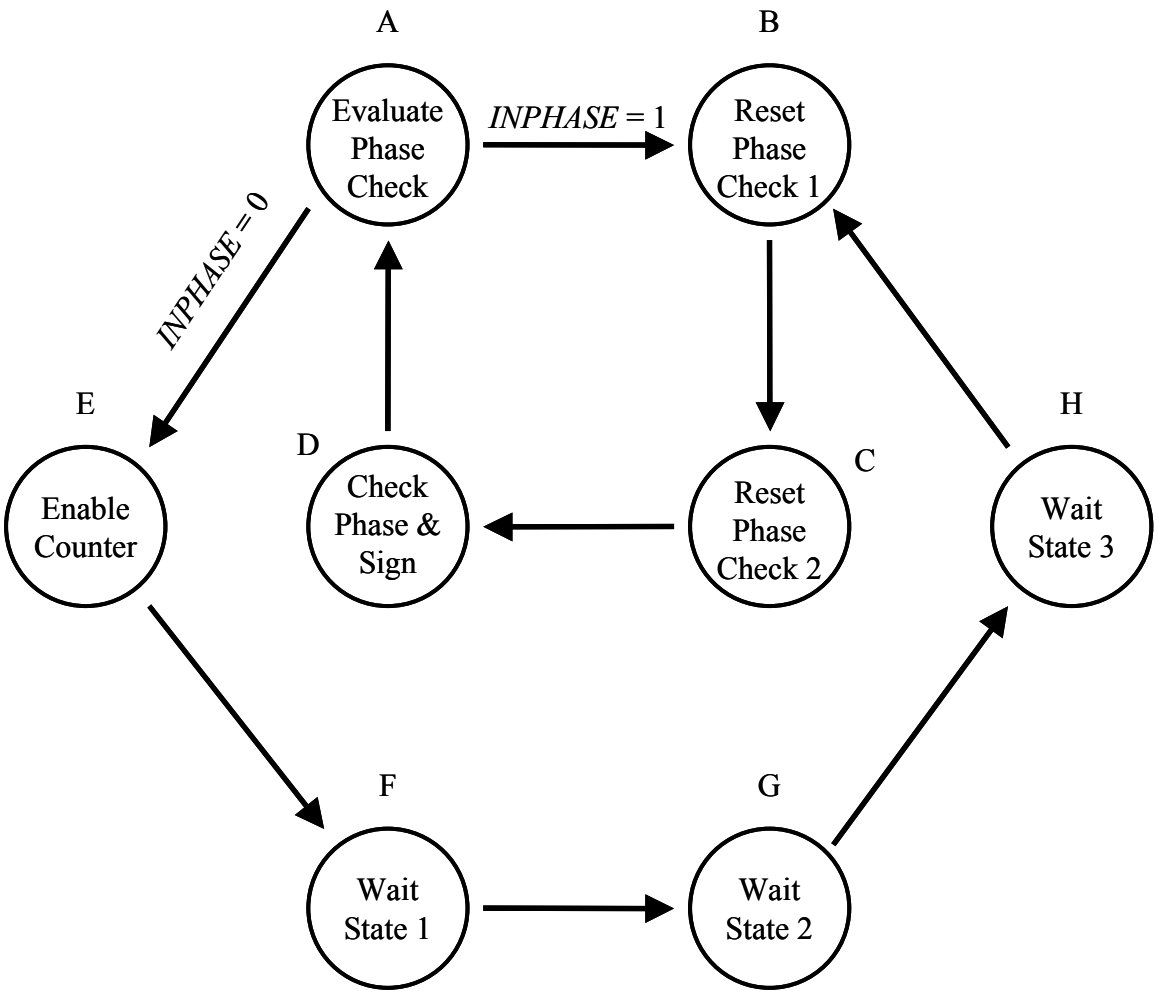

Figure 2.25 State diagram for the finite state machine module.

For simplicity, the FSM outputs are not shown on the state transition diagram. The FSM outputs are listed in the state transition table provided in Figure 2.27. From the state diagram above, in *state A* the FSM evaluates the result of the phase-check module comparison of the *global clock* and C^2 pipeline clock. If INPHASE is asserted (the clock signals synchronous), the FSM transitions to *state B* and then to *state C* while providing a RESET signal to the phase-check module. In *state D*, the reset phase-check module tests the phase of the input clocks and provides the results in the form of the INPHASE and SIGN signals to the FSM. The FSM then transitions back to *state A* and evaluates the phase-check module results. The subset of the state transition diagram where the clocks are synchronized is shown in Figure 2.26. The COUNT signal is negated for the entire cycle when the clocks are synchronized.

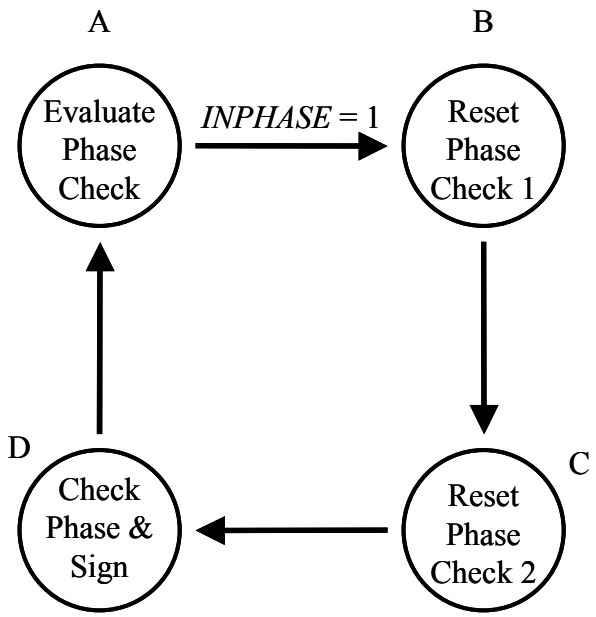

Figure 2.26 Partial FSM state transition diagram (synchronous clocks).

If the clocks are not synchronized, the states transition first around the outer ring then around the inner ring of the state diagram shown in Figure 2.25. Starting again in *state A*, if INPHASE is negated (the clocks are not synchronous), the FSM transitions to *state E* and enables the counter module to increment or decrement in the direction indicated by the SIGN signal from the phase-check module. If SIGN is asserted (i.e., the clocks have positive relative skew), the FSM asserts the UP signal to the counter indicating that an increase in the delay of the pipeline clock is required. If SIGN is negated (i.e., the clocks have negative relative skew), the FSM negates the UP signal indicating to the counter that a decrease in the delay of the pipeline clock is required. The state transition table showing the FSM output signals and corresponding state labels is shown in Figure 2.27.

STATE				NEXT STATE $INPHASE = 0$	NEXT STATE $INPHASE = 1$	OUTPUTS		
q_2	q_1	q_0	Label	Label	Label	COUNT	RESET	HOLD
0	0	1	A	E	B	0	0	0
0	0	0	B	C	C	0	1	X
0	1	0	C	D	D	0	1	X
0	1	1	D	A	A	0	0	1
1	0	1	E	F	F	1	X	X
1	1	1	F	G	G	0	X	X
1	1	0	G	H	H	0	X	X
1	0	0	H	B	B	0	X	X

Figure 2.27 FSM state transition table.

The HOLD signal is internal to the FSM and allows the FSM to latch and hold the SIGN signal from the phase-check circuit during the phase-adjustment portion of the synchronization cycle. Using the state table in Figure 2.27 and Karnaugh maps for logic minimization, the logic equations for the FSM state bits are

$$\begin{aligned} q_0(n+1) &= \overline{q_2}(n)q_1(n) + q_2(n)\overline{q_1}(n)q_0(n) + \overline{q_1}(n)q_0(n)\overline{INPHASE}(n), \\ q_1(n+1) &= \overline{q_2}(n)q_0(n) + q_2(n)\overline{q_0}(n), \\ q_2(n+1) &= q_2(n)q_1(n) + q_2(n)q_0(n) + \overline{q_1}(n)q_0(n)\overline{INPHASE}(n). \end{aligned} \tag{2.2}$$

Similarly, the logic equations for the outputs generated by the FSM are

$$\begin{aligned} \text{COUNT} &= q_2(n)\overline{q_1}(n)q_0(n), \\ \text{RESET} &= \overline{q_0}(n), \\ \text{HOLD} &= q_1(n). \end{aligned} \tag{2.3}$$

A schematic of the FSM circuit is provided in Figure 2.28.

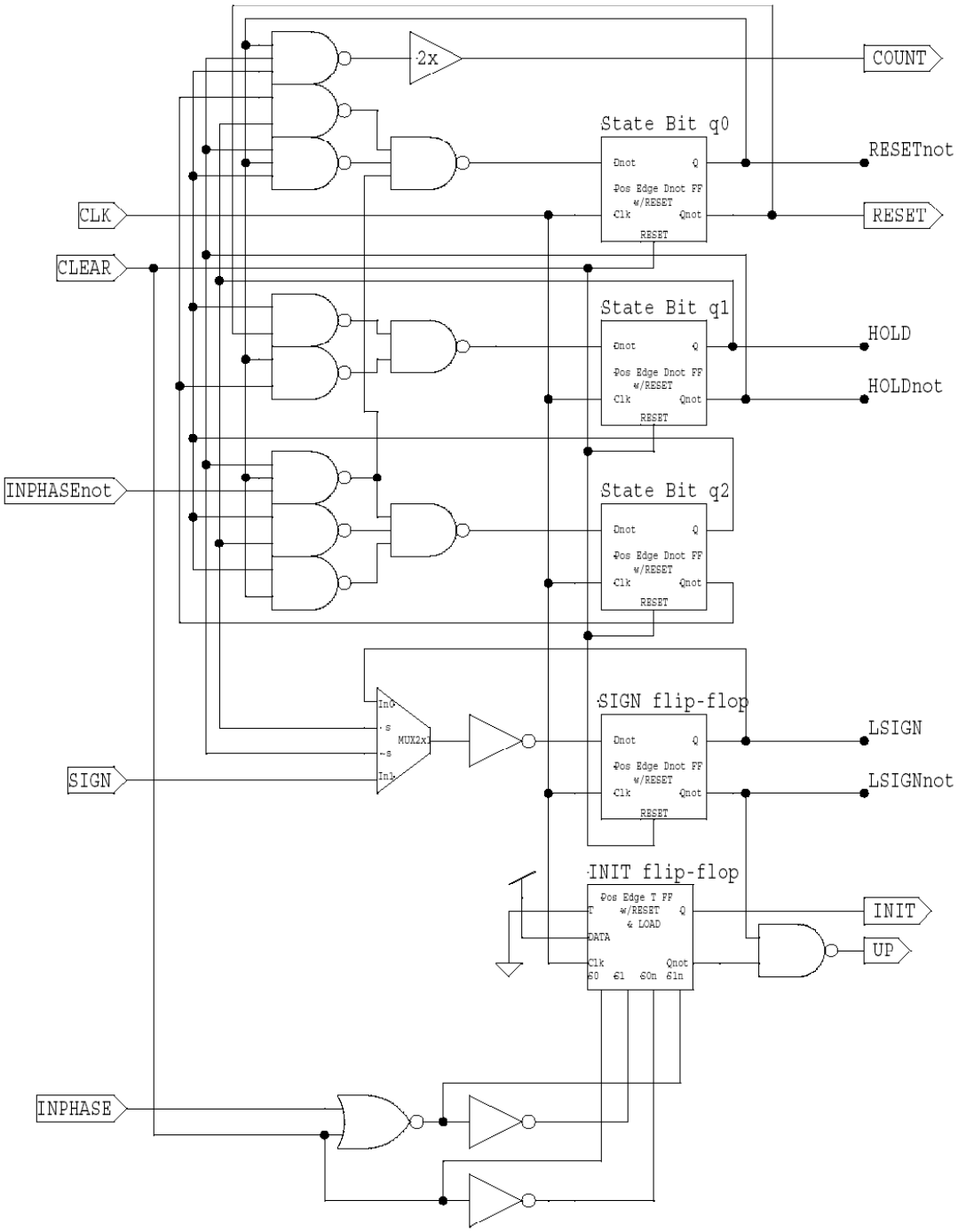

Figure 2.28 FSM Schematic.

The top three D-flip-flops represent the state of the FSM. The fourth flip-flop is a storage element for the SIGN input from the phase-check circuit. The bottom T-flip-flop is a part of the solution to the skew sign ambiguity problem introduced in the previous section. An assumption is made that when the synchronization circuit is initially

activated (i.e., during power-on), the skew between Clock A and Clock B can be any value, including one in which a sign ambiguity exists. However, once the clocks are synchronized, the assumption is that any subsequent clock skew perturbations that result in the clocks losing synchronization will be smaller than the skew that causes a sign ambiguity. For a clock with a 50% duty cycle, this means that there must not be a skew change of $T/2$ in a single synchronization adjustment cycle, nor should the skew change continuously in a single direction faster than the synchronization adjustment cycle can correct. Either of these conditions could potentially lead to a state where the skew sign is ambiguous, which could lead to infinite oscillation in the synchronization adjustment cycle. Therefore, the only problem remaining to be solved is how to deal with two clock signals that have an ambiguous skew sign as an initial condition. The solution implemented in this circuit is to force the counter to only count up when the circuit is initialized, regardless of the state of the SIGN signal from the phase-check circuit, until the clocks are synchronized. Once the clocks are synchronized, the SIGN signal is enabled and the counter can increment or decrement the delay of Clock B according to the value of the SIGN signal. The INIT output of the FSM is asserted anytime the clock synchronization circuit receives the global CLEAR signal, such as during power-on. The INIT signal and counter UP signal remain asserted until the two clock signals are synchronized at which time the INIT output is negated and the UP signal is determined by the sign of the relative clock skew.

c. Wrap-Around Counter

The counter module is essentially a synchronous, loadable and resettable, five-bit parallel counter. A schematic of the core of the counter is shown in Figure 2.29.

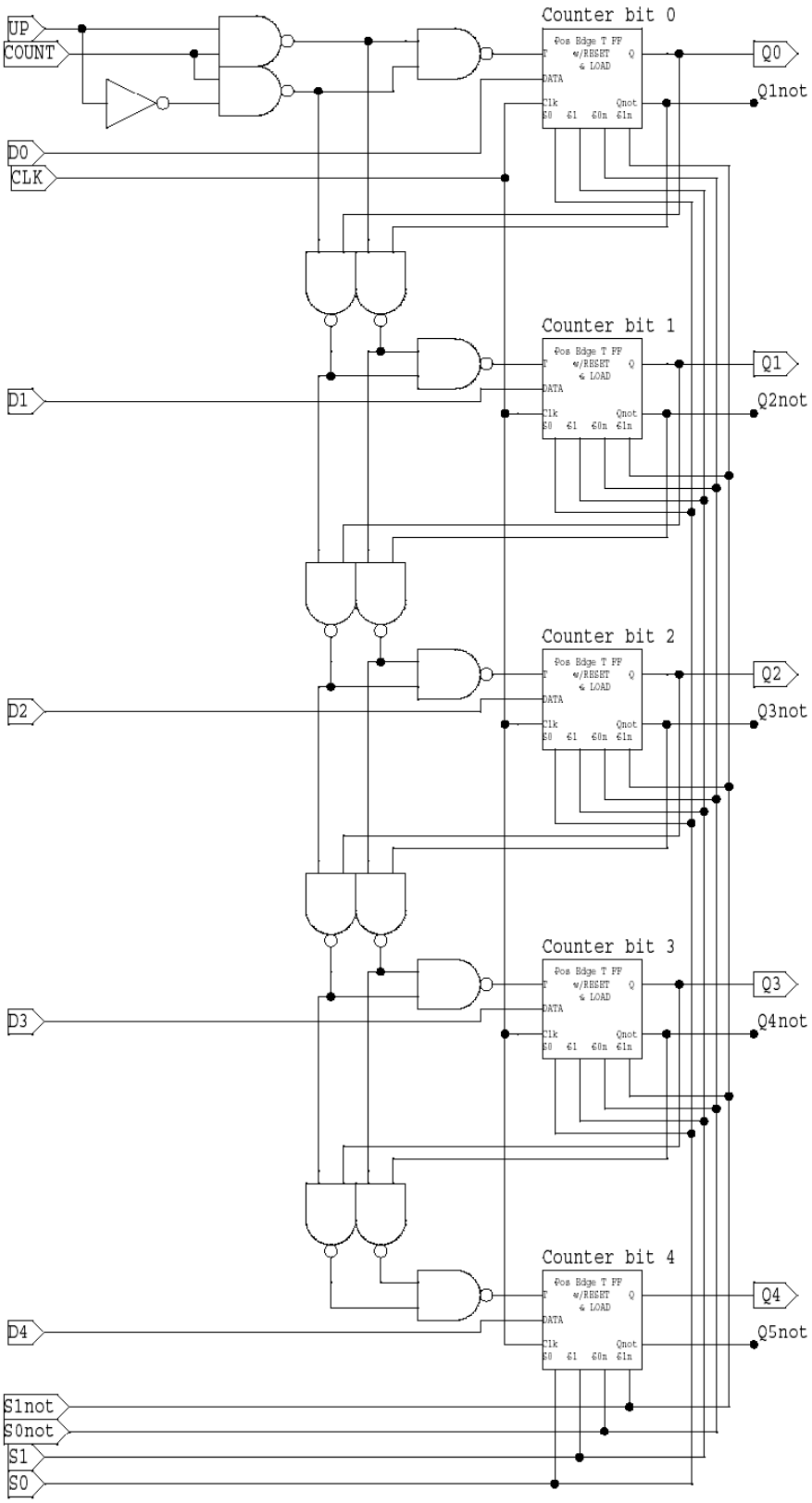

Figure 2.29 Five-bit counter circuit schematic.

The state of the counter is controlled by five T-flip-flops, each with three possible functions: toggle, load, and clear. The particular function performed by the flip-flops is determined by the inputs s_0, and s_1, according to Figure 2.30.

s_1	s_0	Function
0	0	TOGGLE
0	1	LOAD
1	0	CLEAR

Figure 2.30 T-flip-flop functions.

Additional logic must be added to the counter in Figure 2.29 that enables the counter to load a maximum count value and wrap around to that maximum value. For instance, at some point in a synchronization adjustment cycle, it is likely that the counter will be at zero and the FSM will require the counter to count down. Similarly, it is equally possible that the counter will be at its pre-loaded maximum value and be required to count up. The algorithm for the wrap around counter logic is

- Clear the counter when the count equals the maximum count value and the FSM requires a count up or when the CLEAR signal is asserted. T-flip-flops use the *clear* function.
- Load the counter with the maximum count value when the count is zero and the FSM requires a count down. T-flip-flops use the *load* function.
- Otherwise, count normally according to the input signals COUNT and UP from the FSM. T-flip-flops use the *toggle* function.

The logic table for the T-flip-flop function select signals based on the algorithm above is presented in Figure 2.31.

COUNT ASSERTED	CLEAR ASSERTED	COUNT AT ZERO	COUNTING DOWN	COUNT AT MAX	T-flip-flop Function		
co	*cl*	*z*	*dn*	*mx*		s_1	s_0
0	0	0	0	0	TOGGLE	0	0
0	0	0	0	1	TOGGLE	0	0
0	0	0	1	0	TOGGLE	0	0
0	0	0	1	1	TOGGLE	0	0
0	0	1	0	0	TOGGLE	0	0
0	0	1	0	1	TOGGLE	0	0
0	0	1	1	0	TOGGLE	0	0
0	0	1	1	1	X	X	X
0	1	0	0	0	CLEAR	1	0
0	1	0	0	1	CLEAR	1	0
0	1	0	1	0	CLEAR	1	0
0	1	0	1	1	CLEAR	1	0
0	1	1	0	0	CLEAR	1	0
0	1	1	0	1	CLEAR	1	0
0	1	1	1	0	CLEAR	1	0
0	1	1	1	1	X	X	X
1	0	0	0	0	TOGGLE	0	0
1	0	0	0	1	CLEAR	1	0
1	0	0	1	0	TOGGLE	0	0
1	0	0	1	1	TOGGLE	0	0
1	0	1	0	0	TOGGLE	0	0
1	0	1	0	1	TOGGLE	0	0
1	0	1	1	0	LOAD	1	1
1	0	1	1	1	X	X	X
1	1	0	0	0	CLEAR	1	0
1	1	0	0	1	CLEAR	1	0
1	1	0	1	0	CLEAR	1	0
1	1	0	1	1	CLEAR	1	0
1	1	1	0	0	CLEAR	1	0
1	1	1	0	1	CLEAR	1	0
1	1	1	1	0	CLEAR	1	0
1	1	1	1	1	X	X	X

Figure 2.31 Logic table for the T-flip-flop function select signals.

The lower-case italicized signals at the top of the logic table are asserted when the corresponding condition is satisfied. For example, *co* is asserted when the FSM asserts COUNT, *dn* is asserted when the FSM negates UP, *z* is asserted when the all bits of the counter are a logic zero, and so on. Some rows of the table are clearly impossible

in a practical sense, such as a counter reaching its maximum count value and a value of zero simultaneously. Such rows are marked as *don't care* conditions (X). The Karnaugh maps for the wrap-around counter logic table are given in Figure 2.32.

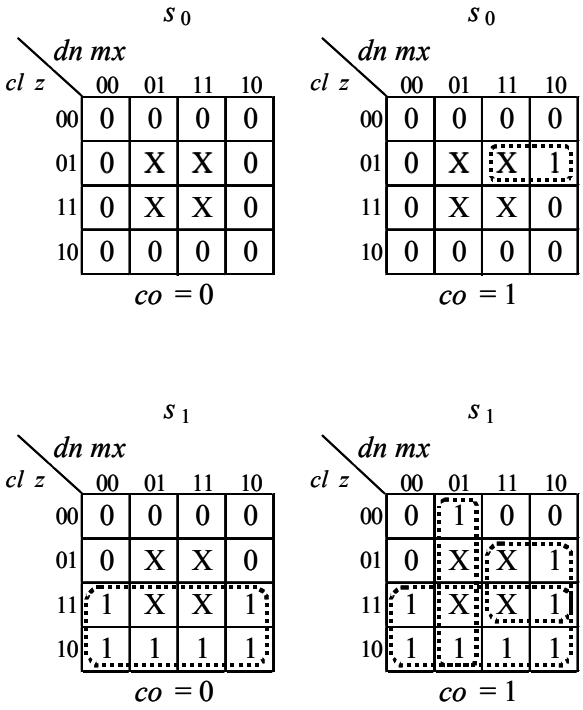

Figure 2.32 Karnaugh maps for the T-flip-flop function select signals in Figure 2.31.

The resulting logic equations are

$$s_1 = cl + co \cdot z \cdot dn + co \cdot \overline{dn} \cdot mx \tag{2.4}$$

and

$$s_0 = \overline{cl} \cdot z \cdot dn \cdot co\,. \tag{2.5}$$

Using (2.4) and (2.5), the schematic for the five-bit wrap-around counter is given in Figure 2.33. The MC0 through MC4 inputs are the binary representation of the maximum counter value. The CLEAR input returns all modules to a known initial state. The UP and COUNT signals are from the FSM, and the Q0 through Q4 outputs are connected to the variable delay module. The state of the signals Q0 through Q4 determines the amount of delay added to Clock B in the variable delay module.

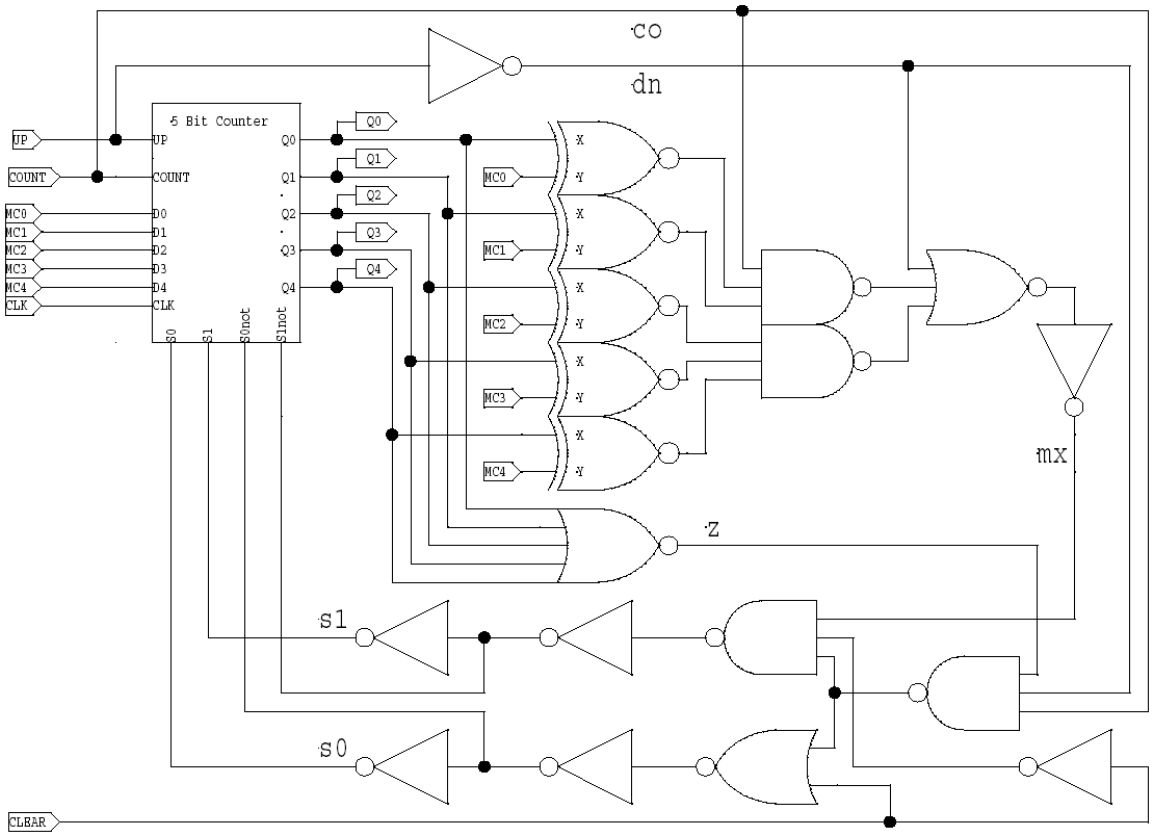

Figure 2.33 Schematic of the five-bit counter with wrap-around logic.

d. Variable Delay Module

The variable delay module is instantiated twice in the clock synchronization design presented in Figure 2.17 and serves two functions – to impart skew on Clock B for testing, and to impart delay on Clock B for synchronization. Consequently, if two clock sources with controllable relative skew are not available for testing the clock synchronization chip, a single clock can be used. The single clock signal represents the *global clock* (Clock A) and is also an input to the first variable delay module, which is used to create a second clock signal with known relative skew (Clock B). The schematic for the variable delay module is shown in Figure 2.34.

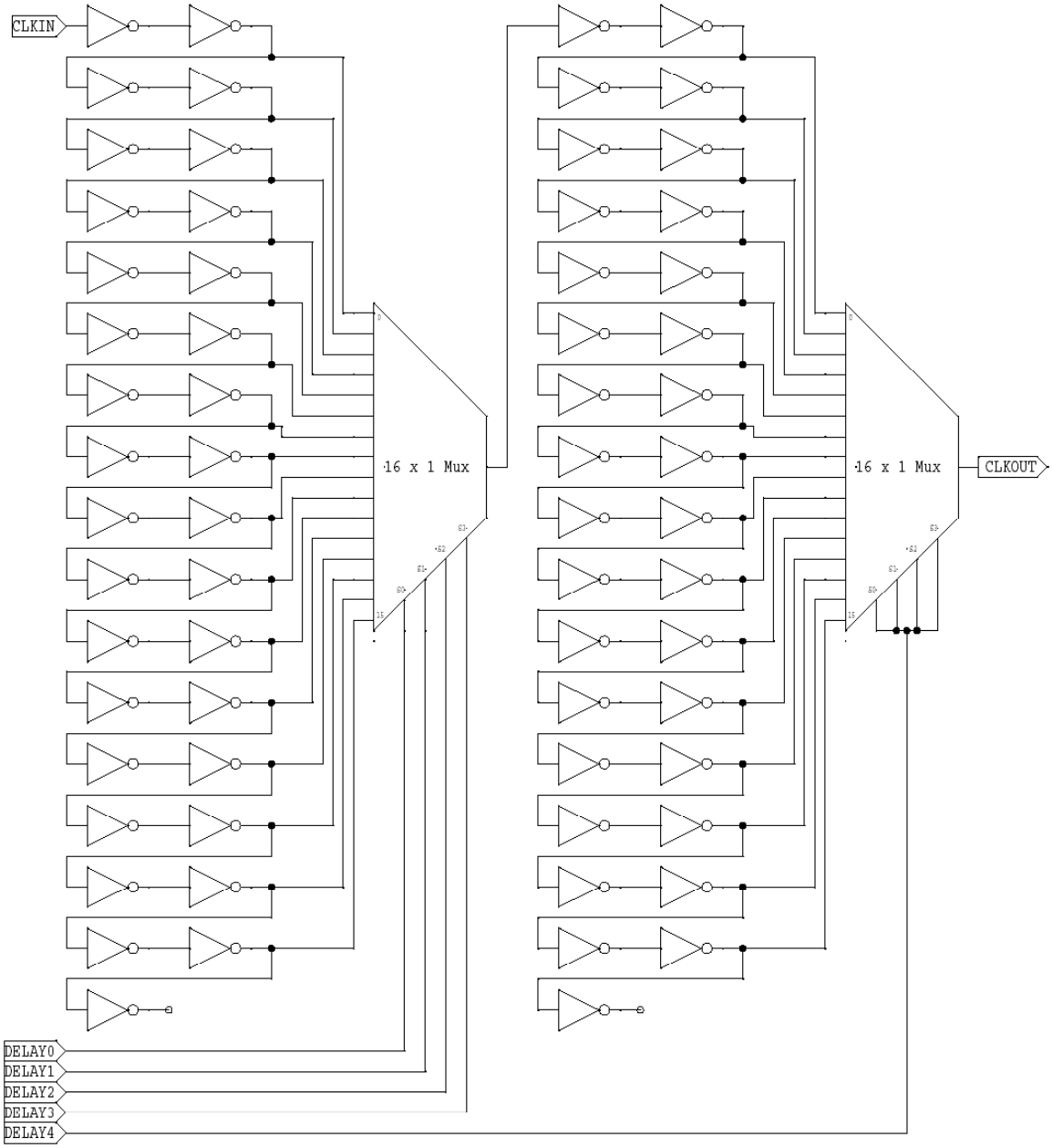

Figure 2.34 Schematic of variable delay module.

A chain of minimum-sized inverters is connected to two 16x1 multiplexers such that there are two inverters between each multiplexer input. The total clock delay at the output of the multiplexer pair is at least

$$2\tau + d_{mux} + 2\tau + d_{mux}, \tag{2.6}$$

where d_{mux} is the delay of the 16x1 multiplexer. A change in the least significant select line causes a relative change in the delay of the output of exactly two minimum-sized in-

verters. Similarly, a change in the next least significant select line causes a relative change in the delay of the output by four minimum-sized inverters, and so on. If the select lines are all zeros, the delay is

$$4\tau + 2d_{mux}. \tag{2.7}$$

If the select lines are all ones, the delay is

$$64\tau + 2d_{mux}, \tag{2.8}$$

which makes the total possible adjustment delay for this circuit

$$\left(64\tau + 2d_{mux}\right) - \left(4\tau + 2d_{mux}\right) = 60\tau. \tag{2.9}$$

Therefore, the maximum correctable skew is 61τ. Using (2.1), this design has the capability to maintain continuous synchronization between two clocks with period

$$T \leq 61\tau. \tag{2.10}$$

For clocks with periods greater than 61τ, this clock synchronization design would still be effective only if the skew was guaranteed to be less than 60τ.

At this point, it is necessary to discuss the computation of the maximum counter value. Ideally, when rolling over the counter from zero to maximum count or from maximum count to zero, the change in the delay of the adjusted clock should still be just 2τ. Figure 2.35 shows a clock signal as well as several incrementally delayed versions of the clock signal.

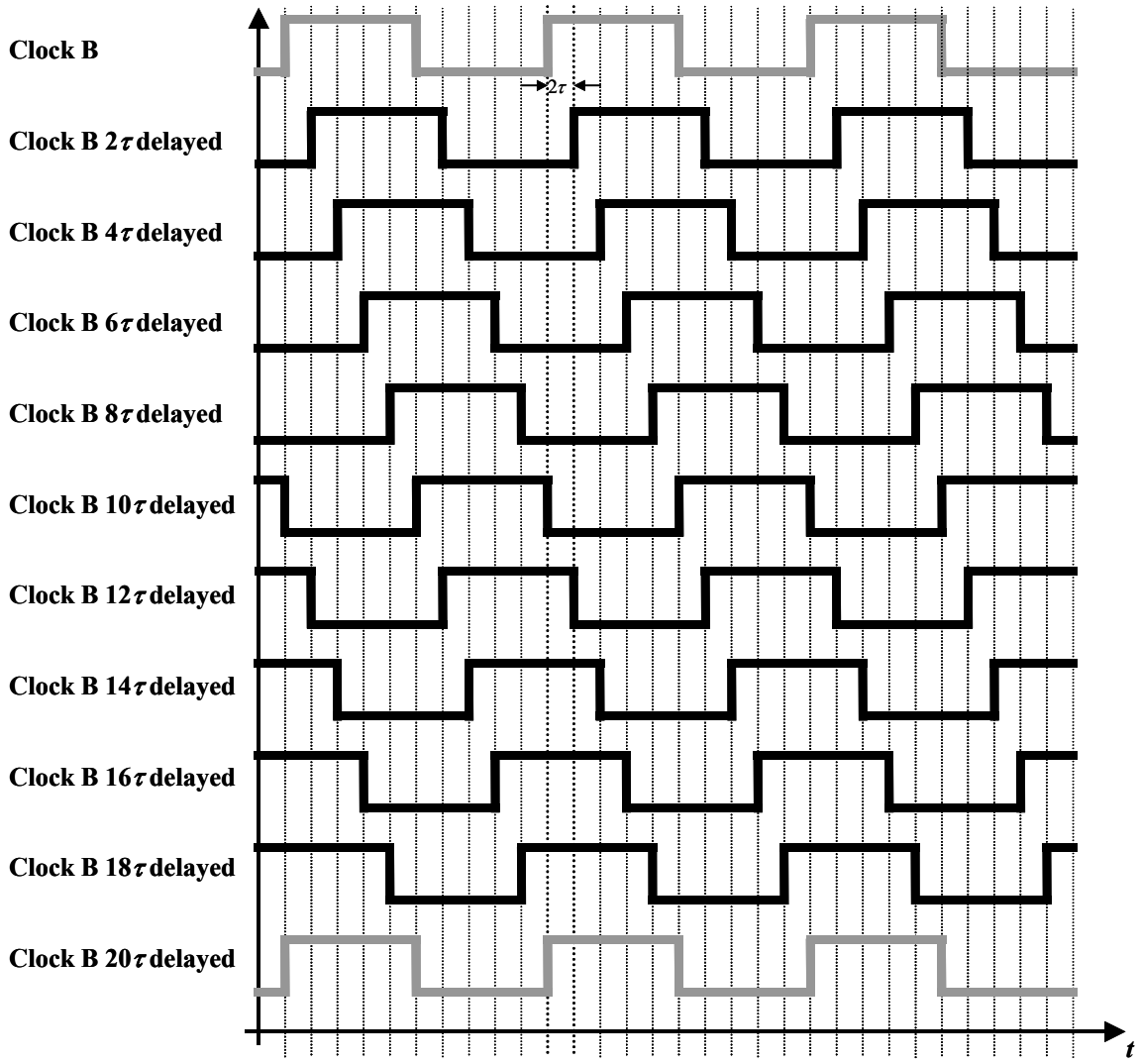

Figure 2.35 Incrementally delayed clock signals.

Notice that Clock B delayed by 20τ has the same phase as the original clock B signal. Thus, for this example, a maximum counter value would be one that produced a delay of 18τ. Consequently, the computation for the maximum count is

$$MC = \lceil (T - 2\tau) / 2\tau \rceil. \tag{2.11}$$

The maximum count is always rounded up so that no infinite oscillations occur due to counter rollover. It would be more conservative to use a maximum counter value of

$$MC = \lfloor T / 2\tau \rfloor, \tag{2.12}$$

which means the delay after rollover is approximately the same. However, a maximum counter value computed using (2.12) could add an additional synchronization cycle to the clock synchronization process.

4. Simulation Results

The clock synchronization chip implementation in this research is composed of approximately two thousand transistors divided among the four major modules. A schematic of the overall design is provided in Figure 2.36.

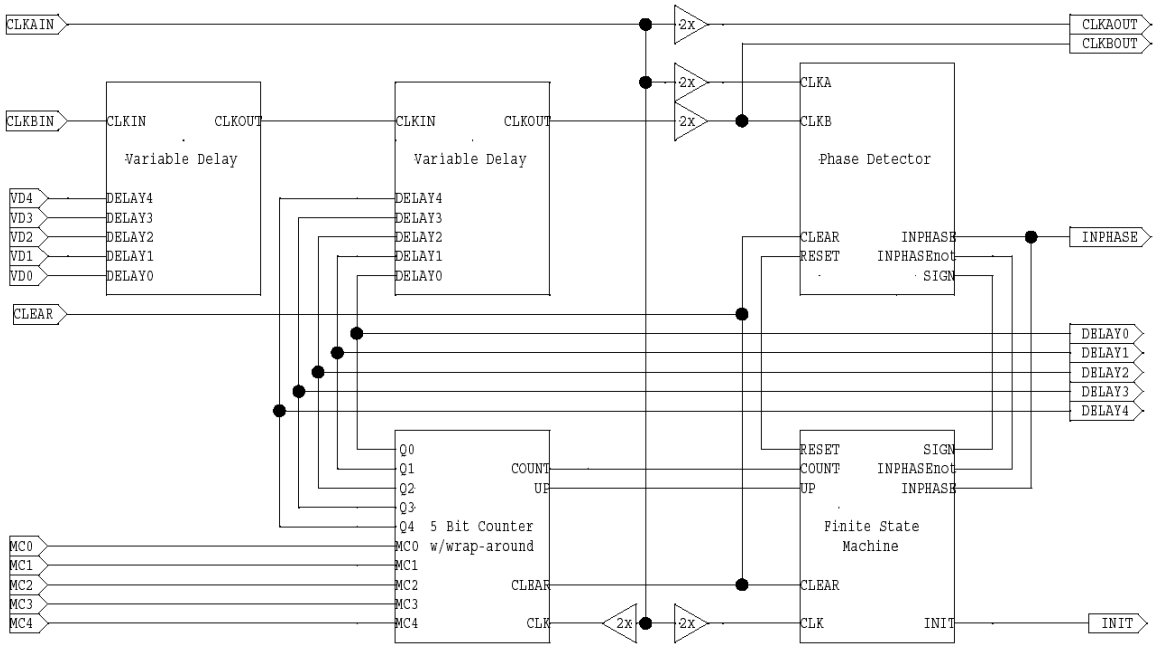

Figure 2.36 Clock synchronization schematic.

Inputs to the circuit are CLKAIN and CLKBIN, the two clock signals to be synchronized. CLKAIN serves as the *global clock* for all synchronous devices in the circuit. Other inputs are VD0 through VD4, which connect to the variable delay module that imposes skew on CLKBIN for testing, MC0 through MC4, which is the binary representation of the maximum counter value, and a global CLEAR signal, which resets the circuit to a known initial state.

The outputs of the circuit are the two synchronized clocks, CLKAOUT and CLKBOUT, the INPHASE signal from the phase detector module, the binary representation of the counter output DELAY0 through DELAY4, and the INIT signal which indicates when the circuit is in the initialization phase. All outputs except CLKBOUT are for chip testing and verification purposes.

All simulations were performed using both Silvaco Parallel SmartSpice version 2.6.0.R and Tanner T-Spice Pro version 7. The simulation results for each schematic were consistent for both versions of SPICE. The simulation results provided in this section are from the Tanner software. The device models and model parameters were provided by MOSIS for the AMI Semiconductor (AMIS) ABN process. The AMIS ABN process has two metal layers, two polysilicon layers and a recommended design lambda of 0.8 micrometers [26]. The model parameters obtained from MOSIS were the average of measured parameters taken from test structures on several wafers in the same lot as the wafer containing the clock synchronization chip.

Simulation results for individual logic gates and minor components in the clock synchronization circuit are provided in Appendix B. Simulation results for all major circuit components are presented in the following sections. For the simulations in the following sections, all input signals are 50% duty cycle, 100-MHz square-wave signals with rise and fall times of 250 picoseconds unless otherwise noted.

a. Phase Check Module

The simulations for the phase check module circuit focused on verifying correct operation of the circuit as well as sizing the transistors in the NAND4 gate to match the average incremental delay of the minimum-sized inverter. Detailed data for the minimum-sized inverter delay is in Appendix B. The average delay of a pair of inverters is approximately 600 picoseconds, previously defined as 2τ. Therefore, to avoid infinite synchronization oscillations, the synchronization threshold must be greater than τ, or 300 picoseconds. The transistors in the NAND4 were sized such that a relative skew of approximately 400 picoseconds triggered the synchronization threshold.

The simulation results provided in Figure 2.37, Figure 2.38, and Figure 2.39 verify the correct operation of the phase check circuit when the clocks are synchronized, unsynchronized with negative skew, and unsynchronized with positive skew respectively.

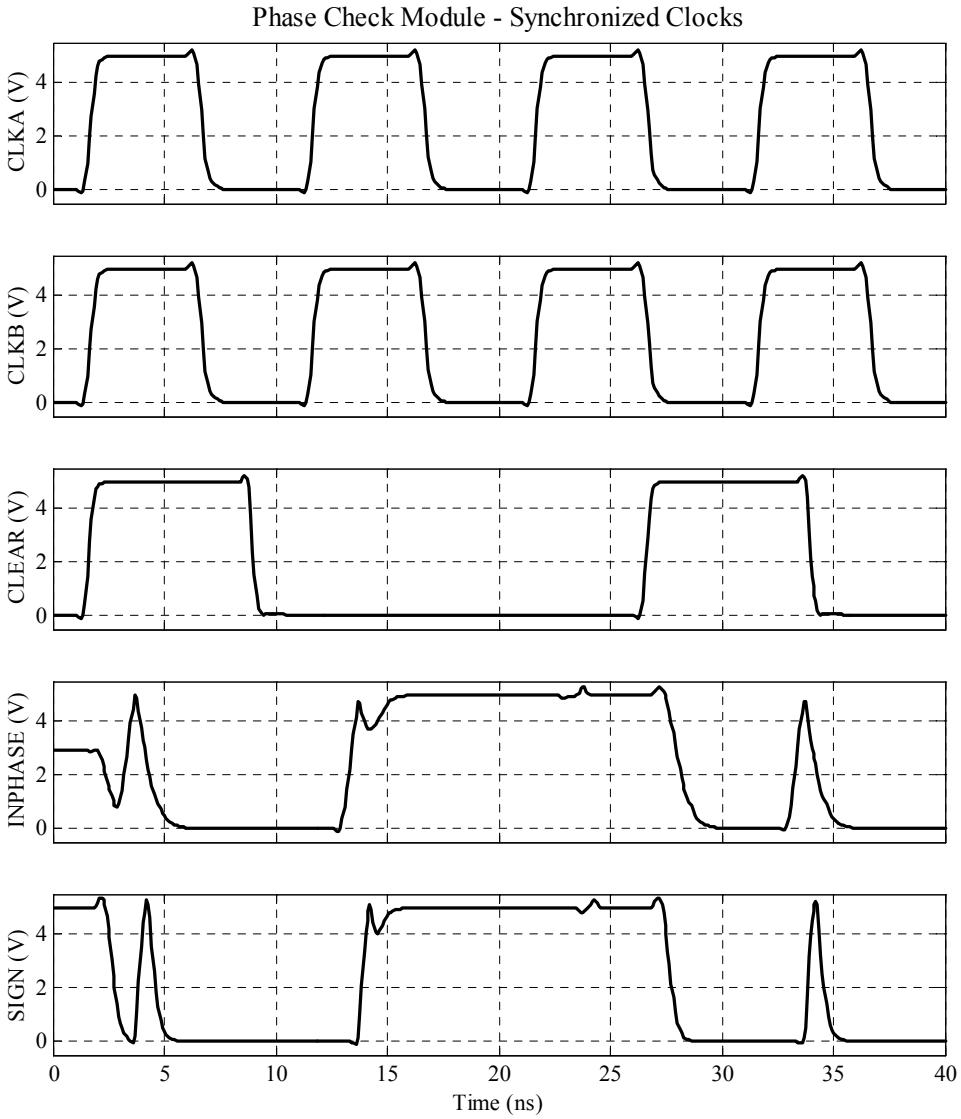

Figure 2.37 Phase-check circuit simulation – synchronized clocks.

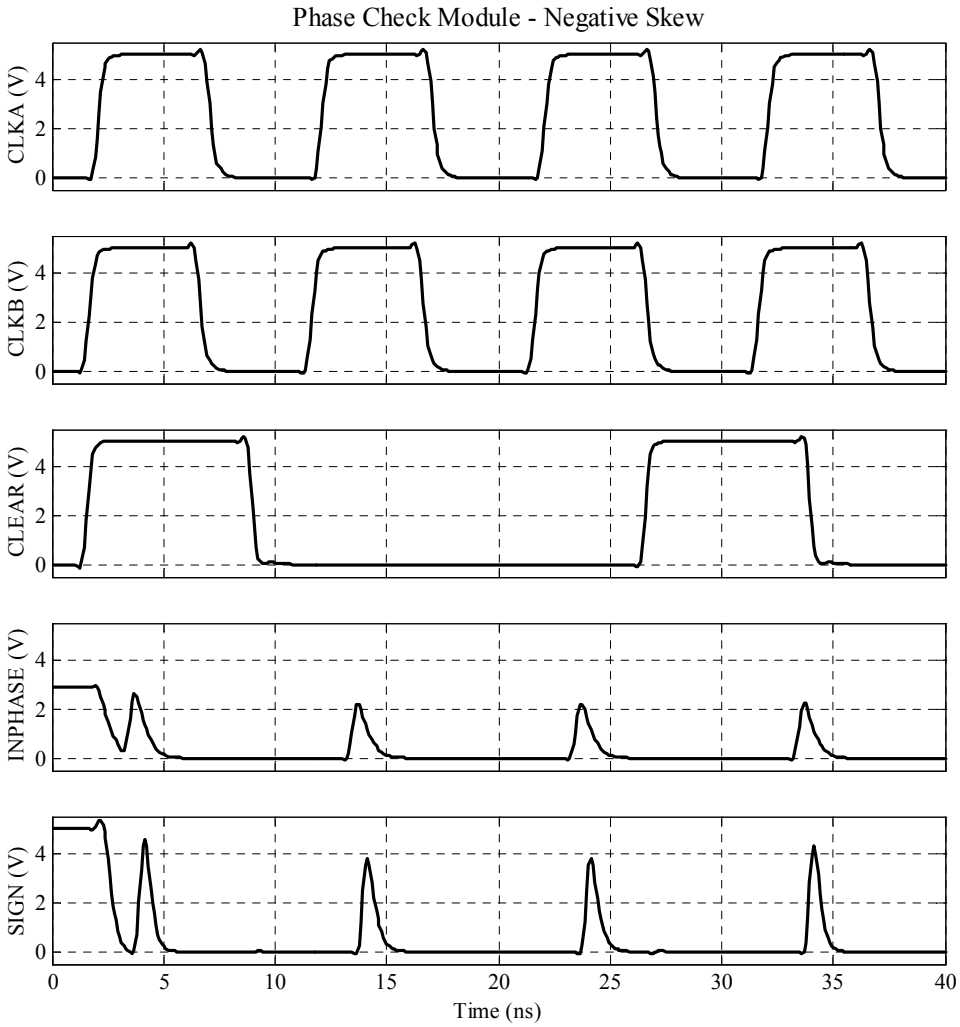

Figure 2.38 Phase-check simulation – unsynchronized clocks with negative skew.

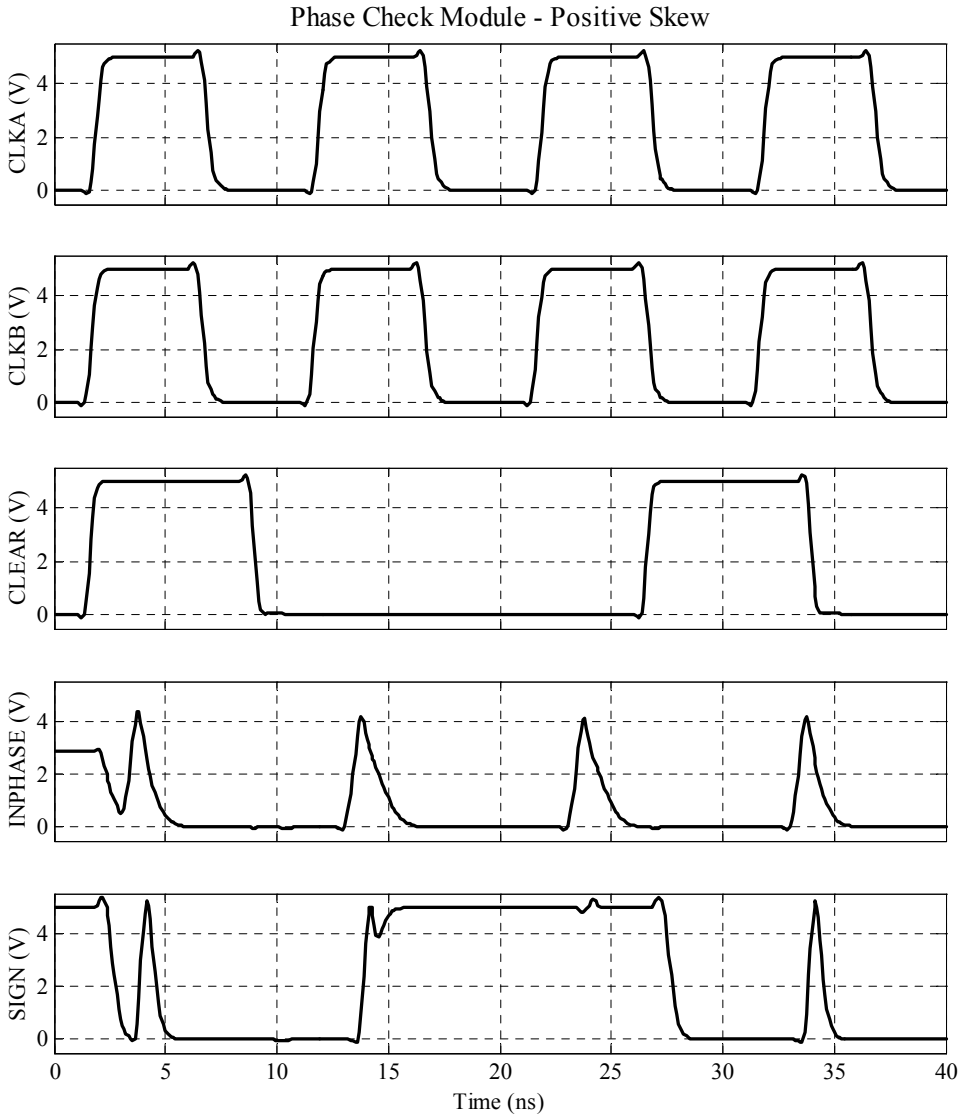

Figure 2.39 Phase-check simulation – unsynchronized clocks with positive skew.

b. Finite State Machine

The simulation results for the FSM are shown in Figure 2.40. In the figure, the INPHASE signal is asserted from 25 nanoseconds (ns) until approximately 150 ns, simulating that the clocks are synchronized. The FSM cycles through states A, B, C, and D as required by the state table in Figure 2.27. At 150 ns, the INPHASE signal changes state simulating that the clocks are no longer synchronized. The FSM cycles through each of the states in Figure 2.27 corresponding to a negated INPHASE signal.

Notice that the COUNT signal is asserted twice at approximately 175 ns and 275 ns, showing two full cycles of phase adjustment.

Figure 2.40 Finite state machine simulation.

c. Wrap-Around Counter

Figure 2.41 shows the simulation results for the five-bit counter. The signals S0 and S1 reset the counter until about 25 ns. At 25 ns, the S0 and S1 signals transitioned to enable the counter load function. An arbitrary value of 27 was loaded into the counter until about the 45 ns point. From 45 ns to just before 100 ns, the counter was allowed to count up from 27 to 31, at which time the counter rolled over to zero and continued to count up. At 200 ns, the UP signal was negated and the counter reversed direction and counted down. At 375 ns, the counter reached zero, rolled over to the maximum counter value of 31, and continued counting down. At 450 ns, the COUNT signal was negated. The counter stopped counting and held its value.

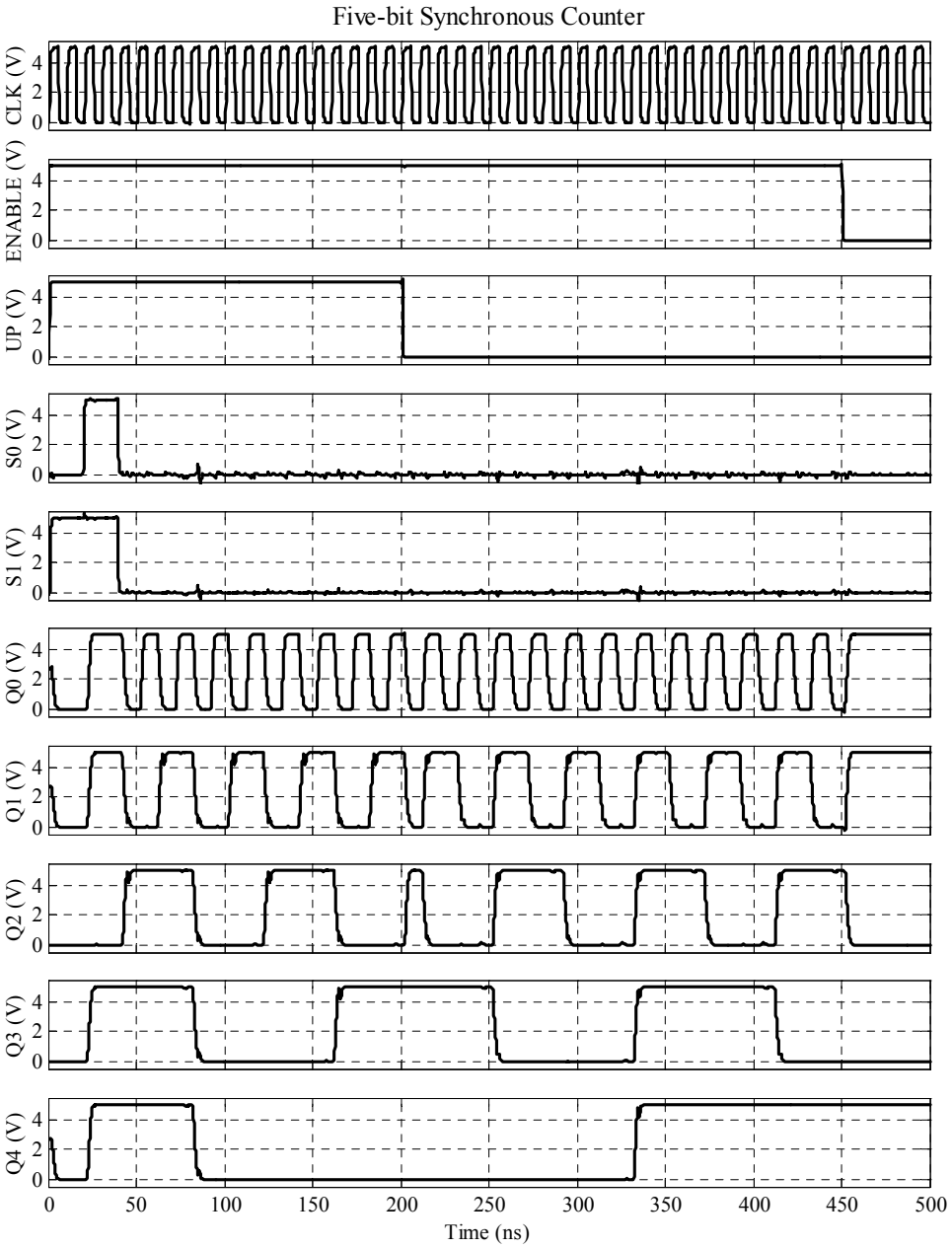

Figure 2.41 Five-bit counter simulation.

Figure 2.42 shows simulation results of testing the five-bit counter with wrap-around circuitry. The RESET signal was asserted until about 25 ns at which time the RESET signal negated and the counter began counting up from zero. At 300 ns, the counter reached the maximum pre-loaded count value of 27, rolled over to zero, and continued to count up. Just as the counter reached a value of 5, the SIGN signal switched states and the counter reversed direction. At just after 400 ns, the counter reached zero,

rolled over to the maximum value of 27, and continued counting down. At 450 ns, the COUNT signal negated causing the counter to stop and hold its value.

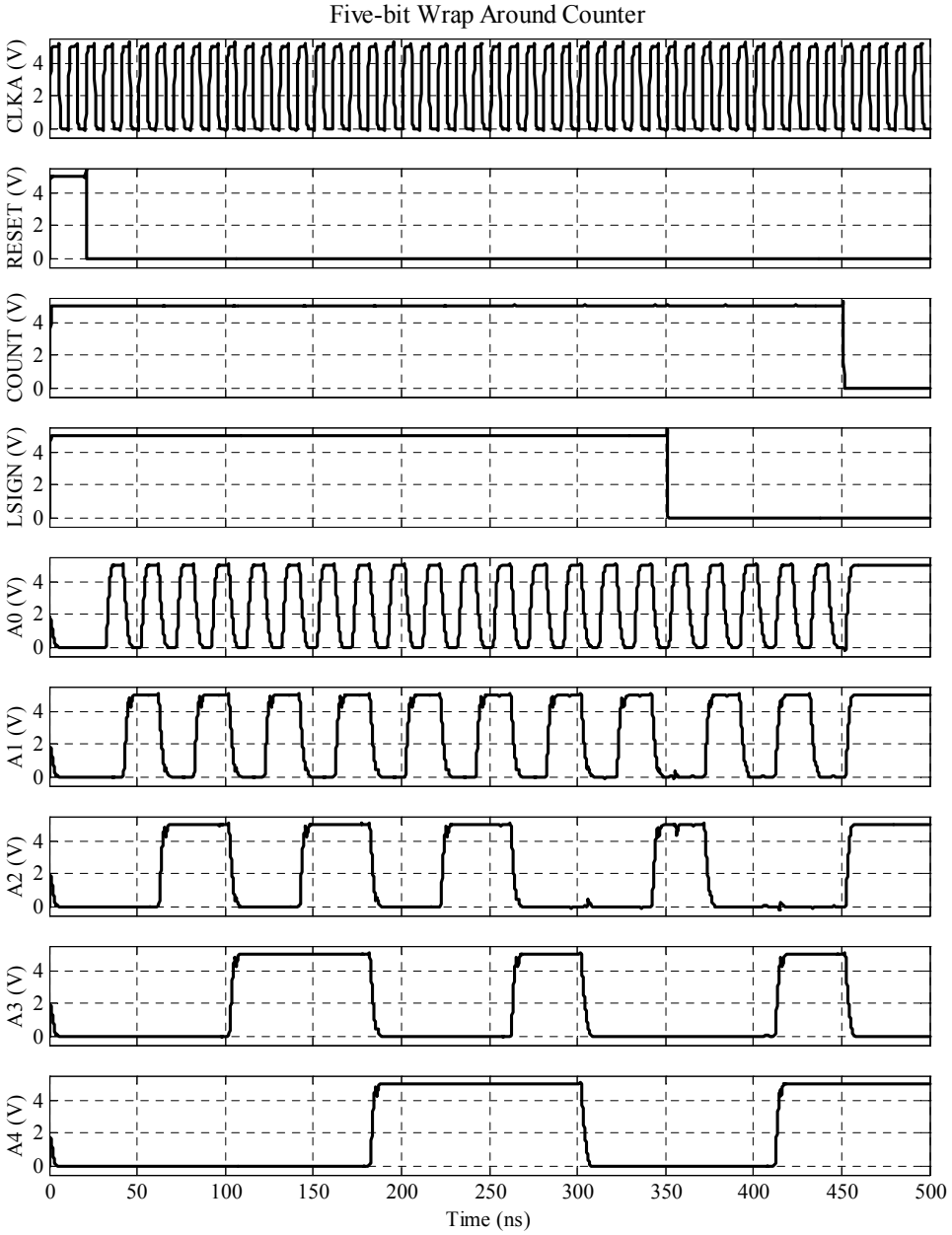

Figure 2.42 Five-bit wrap around counter simulation.

d. Variable Delay Module

Figure 2.43 shows the simulation results for the variable delay module. The top graph shows the clock signal input. The bottom graph shows the output clock signals for four consecutive multiplexer inputs. The relative delay between the output clock signals is roughly a constant and is equal to approximately 600 picoseconds, as expected.

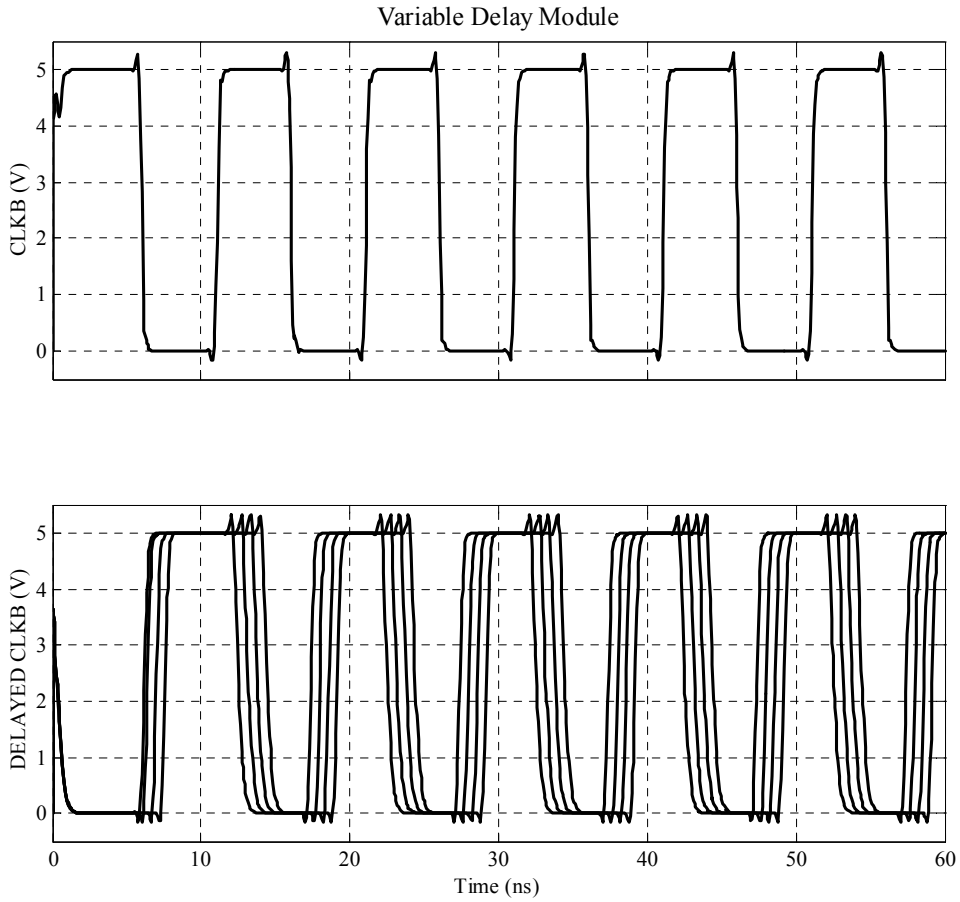

Figure 2.43 Variable delay module simulation.

e. Pad-to-Pad Chip Simulation

Figure 2.44 shows the simulation results of the entire clock synchronization chip and pad ring. The inputs to the circuit are two clock signals with arbitrary rela-

tive skew. The figure shows the INIT signal asserted while the two clocks are initially unsynchronized, then negated once synchronization is achieved.

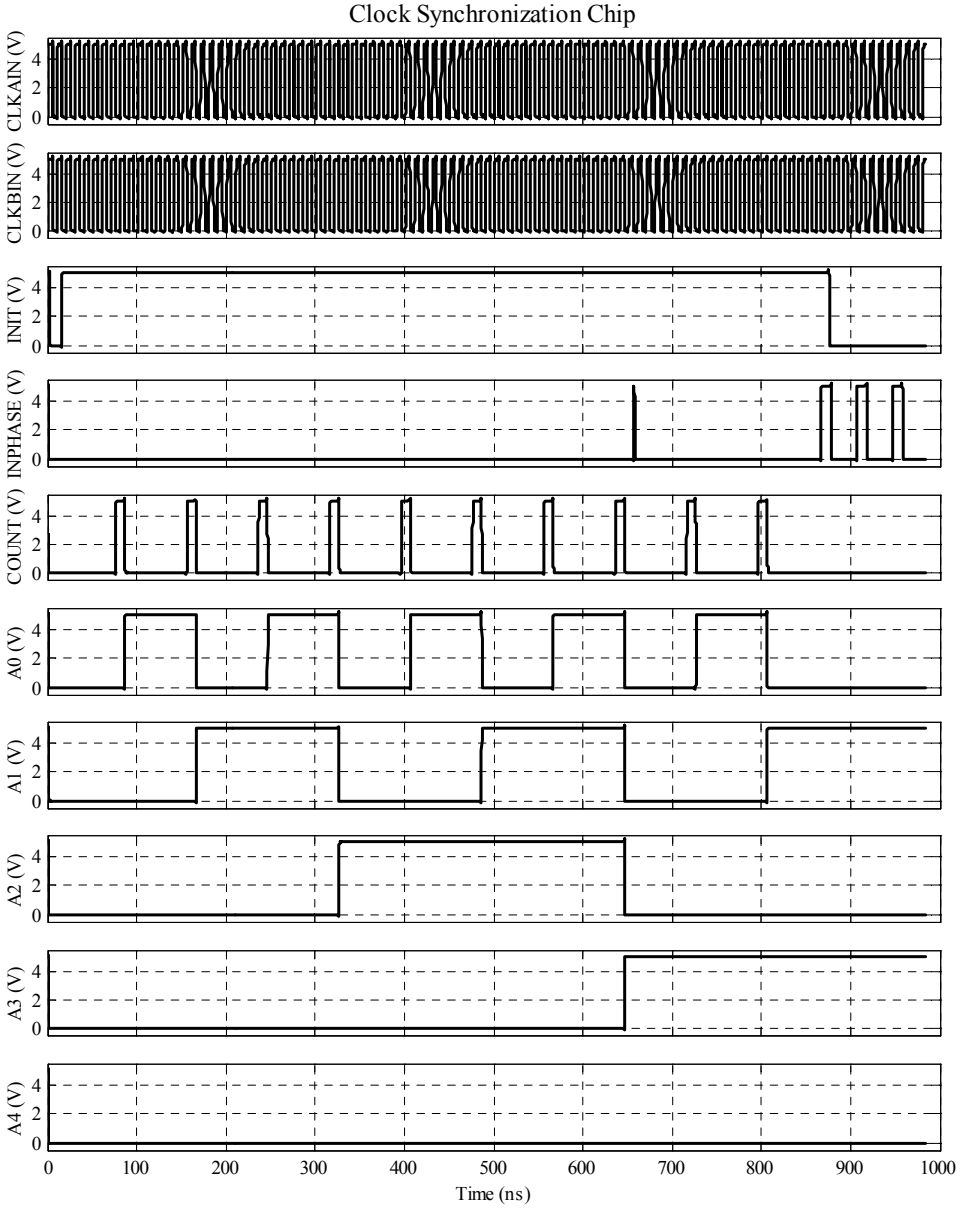

Figure 2.44 Pad-to-pad synchronization chip simulation.

Figure 2.45 shows a close-up view of the clock signals before and after synchronization. Examining the leading edges of the clock signals reveals that there is negligible skew after synchronization.

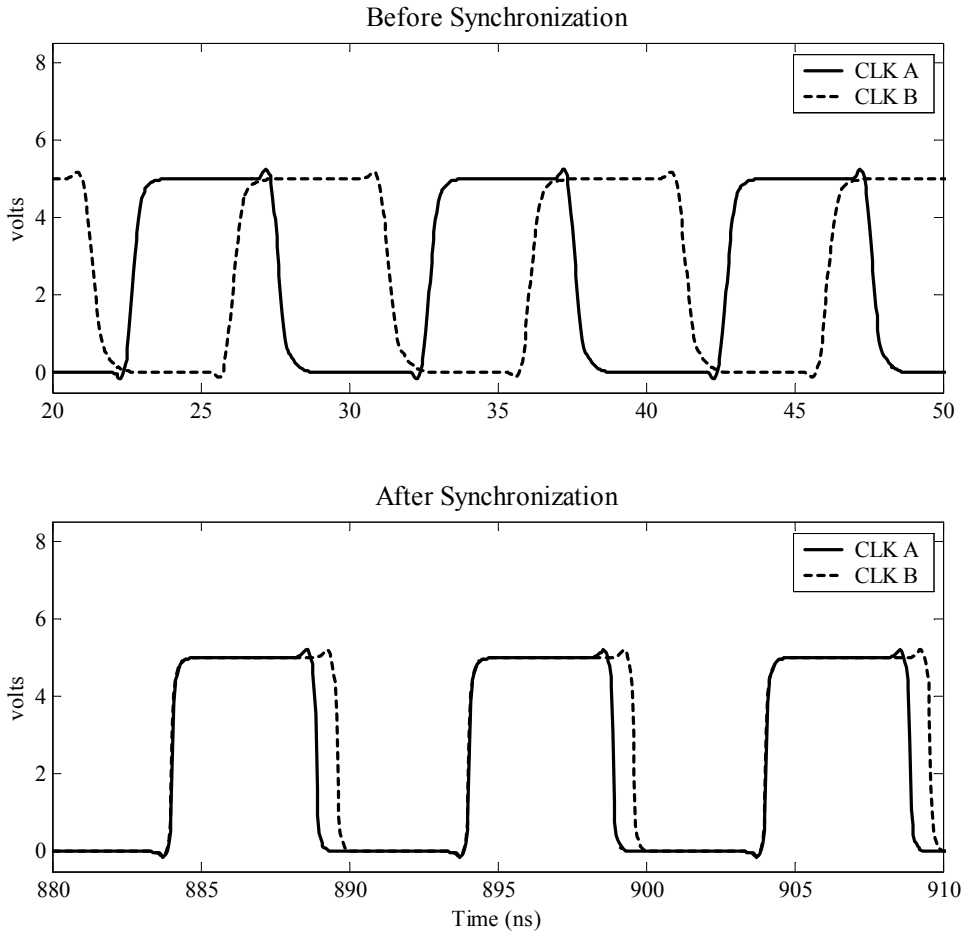

Figure 2.45 Clock signals before and after synchronization.

The same simulation was repeated using nominal fabrication parameters for all values of relative skew from 0 ns to 10 ns in increments of 200 picoseconds (ps). This encompassed all possible skew values for the 100 MHz clock signals. The residual skew remaining after synchronization for each simulation was recorded and is shown in the top graph of Figure 2.46. Chip manufacturers traditionally use the following three terms to describe the boundary performance of transistors [27]:

- Nominal
- Fast
- Slow

Since there are two types of transistors in Complementary Metal Oxide Semiconductor (CMOS) processes, n-type and p-type, the four boundary conditions for the process can be listed as the following:

- Fast-n fast-p (fastfast)
- Fast-n slow-p (fastslow)
- Slow-n slow-p (slowslow)
- Slow-n fast-p (slowfast)

To provide better assurance of correct circuit operation after fabrication, the full-chip simulations run with nominal parameters were also run using the four sets of boundary fabrication parameters listed above. The model parameters for all four boundary conditions for the AMIS ABN process were obtained from the MOSIS website [26]. The residual skew for each of the simulation runs is provided in the bottom four plots of Figure 2.46.

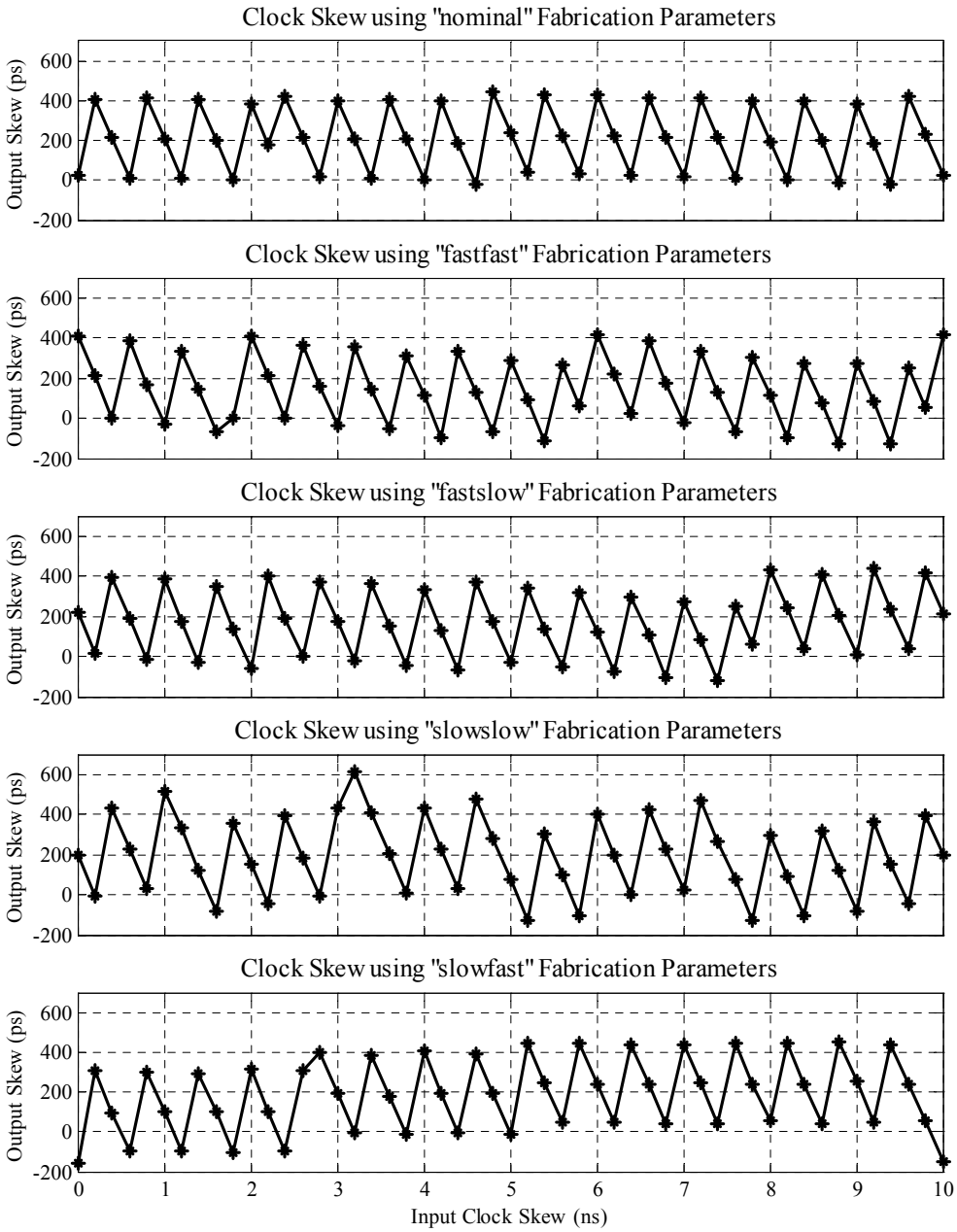

Figure 2.46 Residual clock skew for nominal and four-corners fabrication parameters.

The figure shows that the residual skew between the synchronized clocks ranged from about +400 ps to –100 ps. Since the Variable Delay circuit has an incremental delay resolution of approximately 600 ps, then the results of the simulation indicate that the chip is functioning as intended. Furthermore, since a skew of 10% or less is

normally considered acceptable ([21], [23], [24]), then an average residual skew of 400 ps, which represents a mere 4% of the clock period, is an excellent result.

Additional simulations were run that tested the ability of the synchronization circuit to resynchronize the clocks sometime after initial synchronization. This case represents additional clock skew possibly due to a change in chip operating temperature or power supply voltage, or component aging. For each simulation, the two clocks were allowed to synchronize. Then, additional delay was introduced onto Clock B during the simulation using the variable delay module. In each instance, the circuit automatically re-synchronized Clock A and Clock B as expected.

5. Fabrication and Testing

A proof-of-concept chip implementation of the dynamic clock synchronization scheme described in the previous sections was designed and fabricated using the American Microsystems Incorporated Semiconductor (AMIS) 5-volt, 1.5 micron scaleable CMOS process available through MOSIS. Chip layout is provided in Appendix A. A photograph of the center of the 40-pin ceramic dual-inline package (DIP) showing the 2.2 mm by 2.2 mm square chip and bond wires is provided in Figure 2.47. A pin-out diagram of the chip using the labels from the schematic in Figure 2.36 is provided in Figure 2.48.

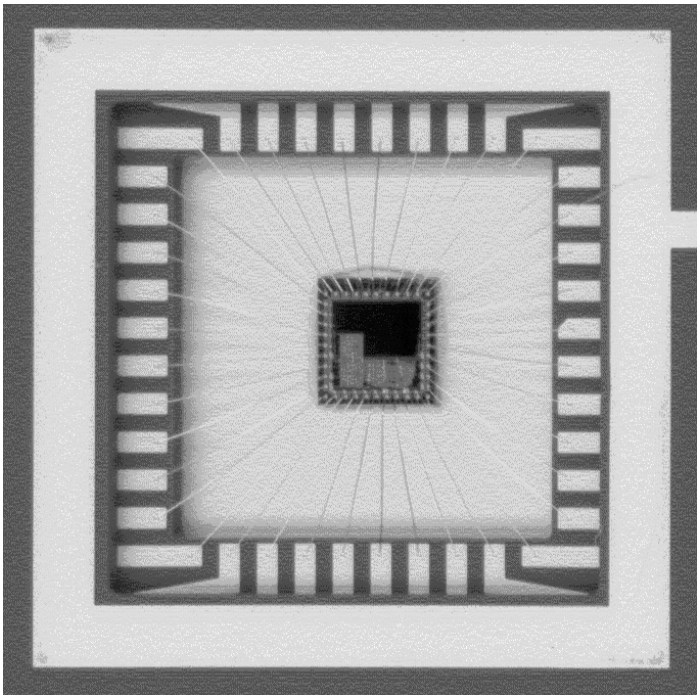

Figure 2.47 Photograph of fabricated chip.

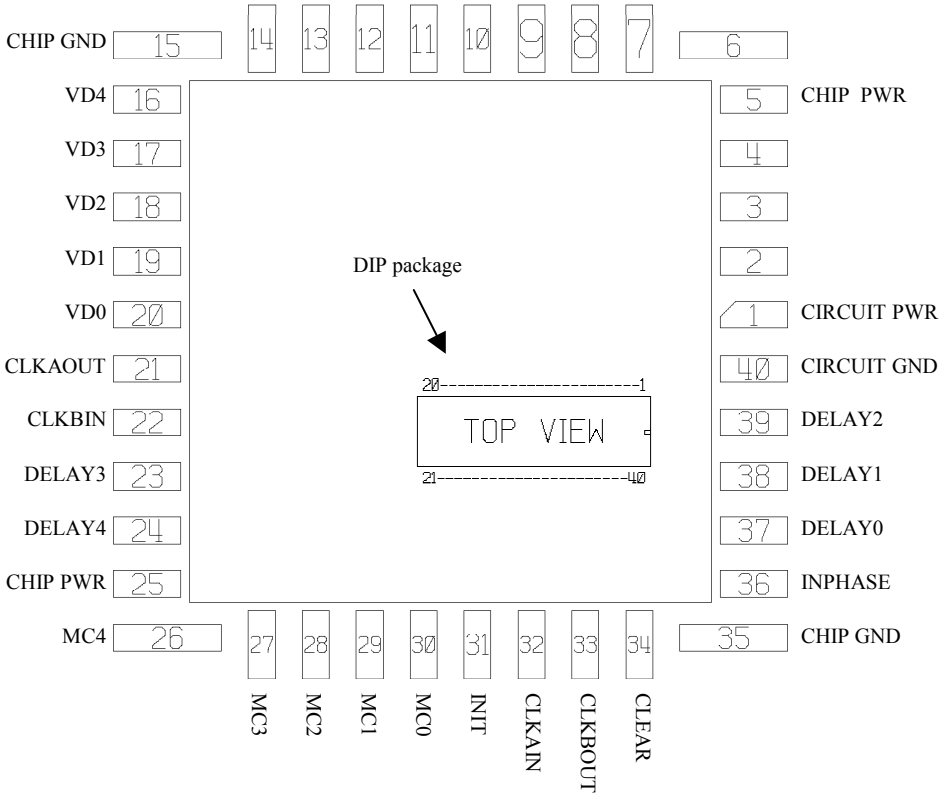

Figure 2.48 Pinout diagram of the synchronization chip.

The test chip DIP was inserted into a wire-wrapped socket on a project board as shown in Figure 2.49. Figure 2.50 provides a picture of the testing setup. A pulse generator provided the Clock A signal to the chip as well as a trigger signal to a second pulse generator. The second pulse generator added programmable skew to the Clock A signal, thereby producing the Clock B signal. Both pulse generators were capable of producing pulses at frequencies up to 500 MHz. The synchronization chip output was captured on a Tektronics TDS 640A 4-channel, 500-MHz digitizing oscilloscope. The chip was tested using 10-MHz square-wave clock signals. Since the period of the input clock signals was greater than the maximum delay possible in the variable delay module, the maximum count was set to the highest possible value.

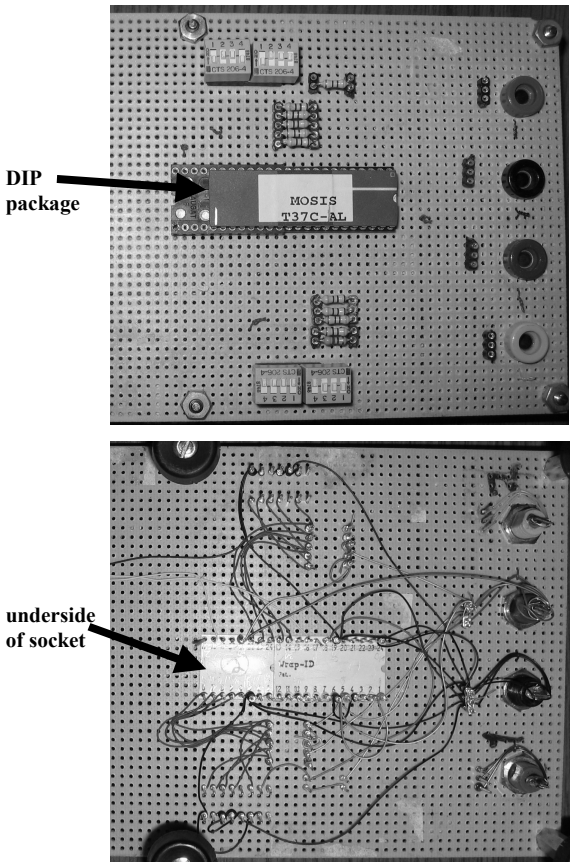

Figure 2.49 Chip testing circuit board from above (top) and below (bottom).

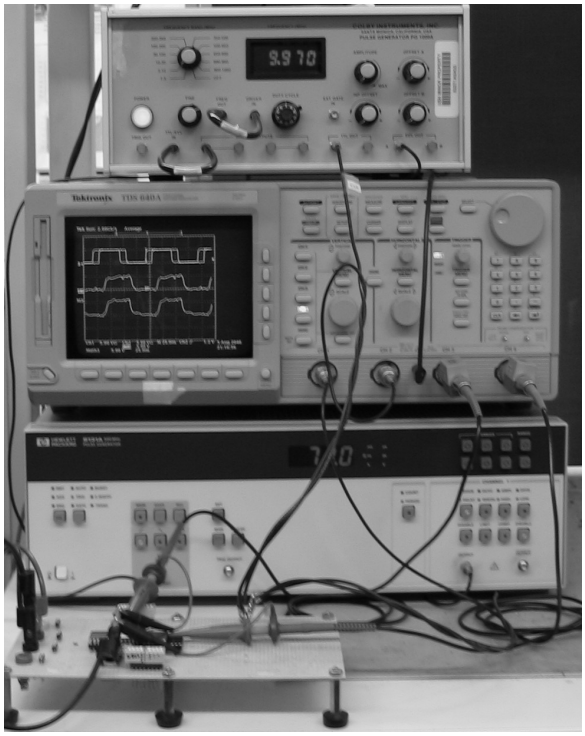

Figure 2.50 Chip testing setup.

Figure 2.51 shows an image from the oscilloscope where the two input clocks (Clock A and Clock B) were perfectly synchronized. Not surprisingly, the two output clocks were also synchronized. The distorted shape of the output signals is due to the fact that the chip output drivers are not sufficient to effectively drive the capacitive load of the digitizing oscilloscope.

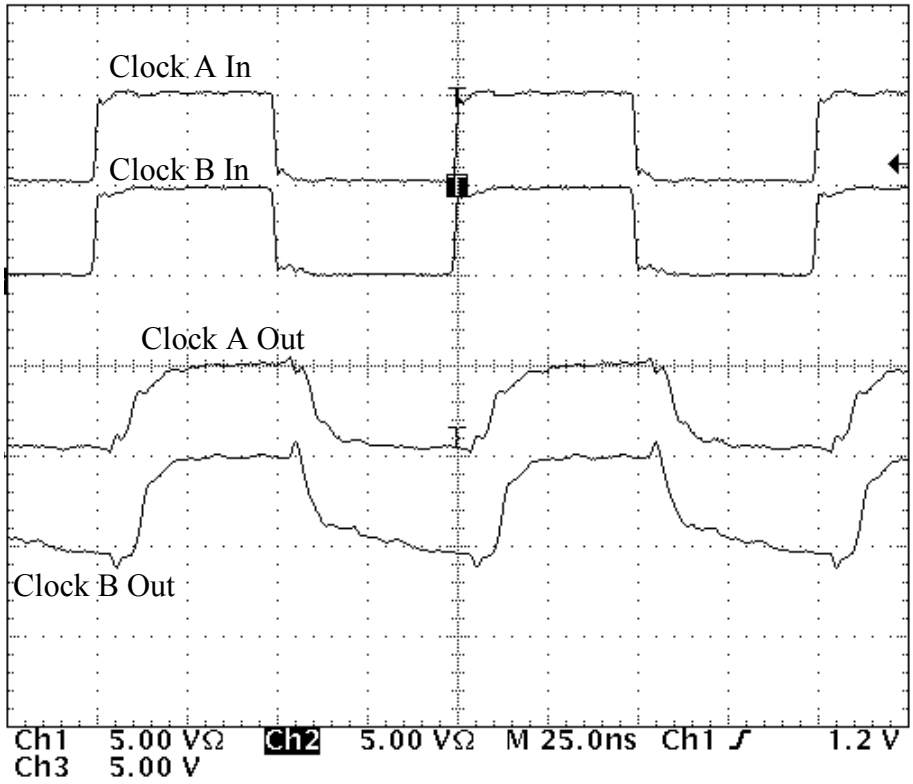

Figure 2.51 Oscillograph of synchronous clocks.

Figure 2.52 shows an image from the oscilloscope where the two input clocks (Clock A and Clock B) were out of phase by almost 25 nanoseconds. The figure shows that the output clocks are synchronized despite the skew present between the input clocks.

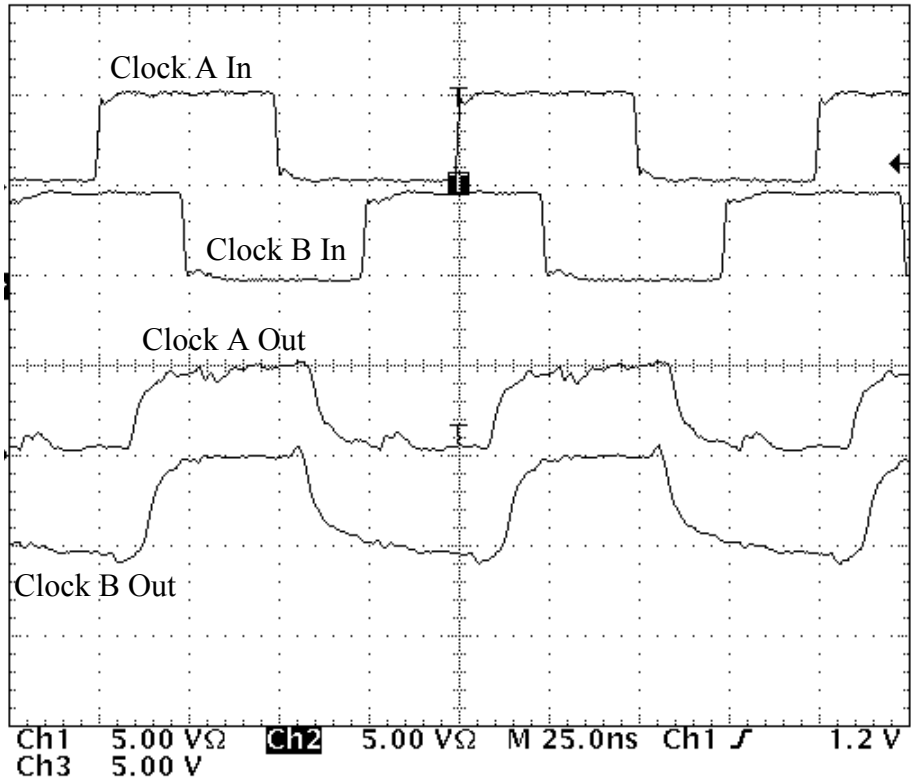

Figure 2.52 Oscillograph of synchronized clocks.

Figure 2.53 shows an image from the oscilloscope where the two input clocks (Clock A and Clock B) were out of phase by more than 30 nanoseconds. In this instance, the two output clocks are not able to synchronize due to the fact that there is not enough delay in the variable delay module to correct the skew. The oscilloscope image shows the Clock B signal even more distorted than the previous synchronized images. This is because the Clock B signal is rapidly searching for phase lock while the variable delay module continually changes the delay on the signal. In order to comprehensively test the circuit, the relative skew was set to zero and increased in increments of 100 picoseconds until the circuit could no longer synchronize the two clock signals.

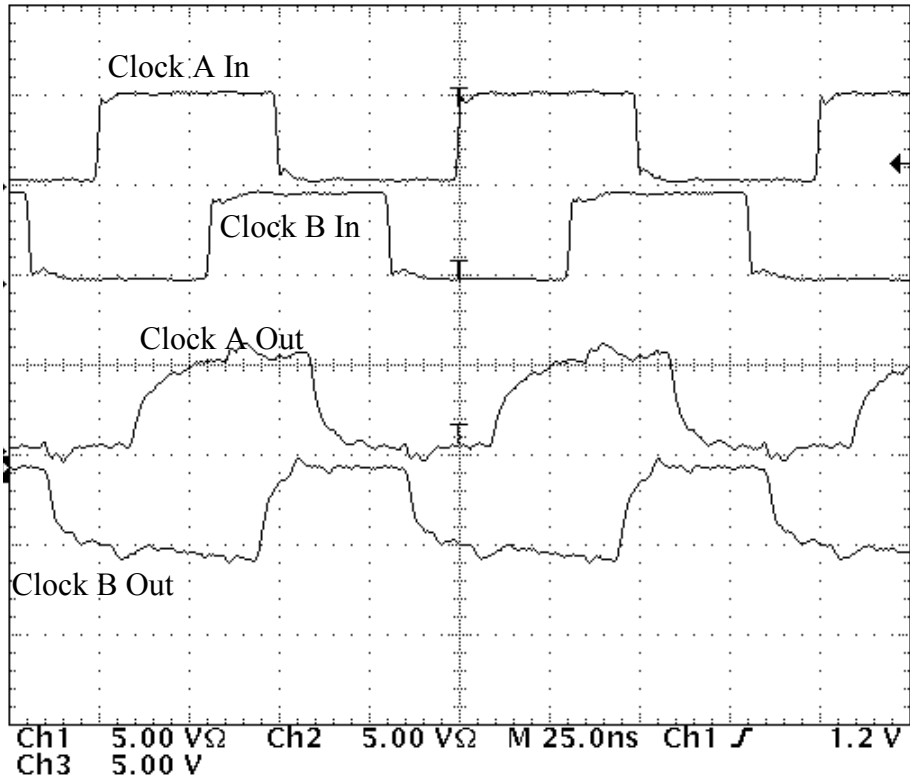

Figure 2.53 Oscillograph of unsynchronized clocks.

Figure 2.54 shows the predicted and measured results for four different maximum count values. The predicted synchronization range was based on the simulated incremental delay value of 600 picoseconds. The measured synchronization ranges were larger than the simulations predicted and corresponded to an average equivalent incremental delay of 851 picoseconds. The increase in incremental delay is likely due to variation in fabrication parameters as well as wire capacitances not taken into account in simulation. However, the change in incremental delay merely extended the capture range of the synchronization circuit and did not adversely affect the ability of the circuit to achieve synchronization.

Max Count	Predicted Skew Synchronization Range	Measured Skew Synchronization Range	Equivalent Incremental Delay
31	18 ns	25.8 ns	860 ns
15	9 ns	12.7 ns	846 ns
11	6.75 ns	9.1 ns	871 ns
7	4.5 ns	6.1 ns	827 ns

Figure 2.54 Table of predicted and measured synchronization ranges.

D. SUMMARY

Although the current DIS architecture contains at least two sources of redundant computation, no alternative architectures were found which produced a reduced transistor count that was compatible with the DIS fabrication process. The alternative architecture utilizing the *I/Q* high-speed bus distribution design is promising in terms of reducing transistor count, but the DIS would have to be fabricated using another process that would allow for a six-times increase in the speed of the incrementer.

Since simulating an entire SoC is prohibitively time consuming if possible at all, each SoC component must be reliable regardless of changes to the other components on the chip. The clock synchronization design presented in this chapter was proven to be an effective method for automatically controlling the clock skew inherent in counterflow-clock pipelined circuits. Use of the clock synchronization architecture in the current DIS design would render manual synchronization adjustments unnecessary. Furthermore, the clock synchronization technique provides continuous automatic synchronization of off-chip and counterflow clock signals due to clock skew caused by unanticipated fabrication process variations or dynamic changes in chip operating characteristics. The advantages of this design over other dynamic synchronization techniques are that this design uses only digital components and uses less power than phase-locked loop based designs [24]. Moreover, all components except the phase-check NAND4 gate are scalable as opposed to analog components that must be completely redesigned with each fabrication process change.

The next chapter addresses the analog-to-digital converter design for the EW SoC. Similar to the DIS and the clock synchronization circuit, the primary design constraints for the ADC were low-power operation and compact layout. The ADC design marks a transition from the all-digital components presented in this chapter to analog and digital components requiring mixed-signal design techniques.

III. RSNS ANALOG-TO-DIGITAL CONVERTER

One of the goals of this dissertation was to advance mixed-signal system-on-a-chip technology to the point where an entire electronic decoy system can be implemented on a single chip. To accomplish this goal, the ADC must have a small die footprint and operate at extremely low power. However, the ADC must maintain a high sample rate. Two of the fastest ADC architectures in use today are the flash and folding architectures [28], [29]. The characteristics of RSNS folding ADCs provide benefits in the areas of conversion speed, power savings, and die area. Folding ADC architectures offer the most resolution at high speeds of any existing ADC architecture [29]. Compared to traditional ADC designs, the number of comparators is minimized using the RSNS design and the interpolation circuitry can be eliminated completely [30]. This reduces both the die footprint and power consumption.

The ADC is the most fundamental building block in a signal processing system. The first ADC presented in this chapter is a six-bit ADC conceptually identical to the design described in [30]. The ADC is a folding ADC that is designed to implement the RSNS in order to realize the advantages mentioned above. In an RSNS folding ADC, the analog input signal is fed into parallel data channels where it is folded into periodic, triangular waveforms associated with unique relatively prime moduli. The first ADC presented in this chapter is a three-channel ADC. The ADC design in [30] was a purely asynchronous analog design and was simulated using generic models for the circuit elements. In order to produce a proof-of-concept hardware implementation of the ADC presented in [30] that includes synchronous CMOS decoding circuitry, the ADC design must be modified and subsequently simulated using realistic circuit element models and parameters for a mixed-signal fabrication process. The first section in this chapter describes a mixed-signal fabrication process for implementation of a proof-of-concept RSNS ADC. The second section presents an RSNS ADC design based on [30] suitable for fabrication in the mixed-signal process.

The second ADC presented in this chapter is a four-channel, eight-bit RSNS ADC. Since the inputs to the DIS described in Chapter II are 8-bit digital I and Q signals,

two 8-bit ADCs are required to convert the analog I and Q signals to digital. Therefore, the second section in this chapter provides just such a circuit and compares the size and power of the eight-bit, four-channel RSNS ADC to an eight-bit, three-channel ADC. Furthermore, the eight-bit ADC illustrates the procedure for designing N-channel k-bit RSNS ADCs. The final section in this chapter provides simulation results for both the three-channel and four-channel RSNS ADCs.

A. SILICON-GERMANIUM MIXED-SIGNAL FABRICATION PROCESS

Due to the low-volume nature of proof-of-concept VLSI fabrication, many prototype chips from academic institutions are fabricated using processes available through MOSIS. Although MOSIS provides access to a few mixed-signal processes, the IBM silicon-germanium (SiGe) process provides a promising mix of analog and complementary metal-oxide semiconductor (CMOS) performance.

The IBM BiCMOS 5HP process was chosen for simulating the ADC circuits in this chapter because it offered adequate analog and digital performance within budget constraints. The IBM BiCMOS 5HP integrates a high-performance trench-isolated SiGe heterojunction bipolar transistor (HBT) with a 3.3-V, 0.5-µm CMOS base. Therefore, in addition to being analog and radio-frequency (RF) friendly, the IBM 5HP process is a true mixed-signal technology. IBM SiGe has higher performance in terms of cutoff and maximum oscillation frequencies (f_t and f_{max}) and higher Early Voltage (Beta-V_A product) than silicon, while also having a lower turn-on voltage, lower power, and lower $1/f$ noise than gallium-arsenide (GaAs) [31]. Furthermore, the cost to fabricate SiGe chips is typically much lower than GaAs because SiGe runs are performed on readily available silicon fabrication process lines. The specifications for the IBM SiGe 5HP process are provided in Figure 3.1.

Specifications

SiGe HBT NPNs

	Small-Signal/Low-Voltage High-Speed Device	**High-Power/High-Voltage Low-Noise Device**
BV_{ceo}	3.3 V	5.5 V
Gain	100	80
f_t	47 GHz	30 GHz
f_{max}	65 GHz @ V_{cb} = 1 V, V_{be} = 0.72 V	55 GHz @ V_{cb} = 1 V, V_{be} = 0.72 V,
V_{early}	65 V	124 V

FETs

	NFET (W/L = 10/0.5 µm)	**PFET (W/L = 10/0.5 µm)**
T_{ox}	7.8 nm	7.8 nm
L_{eff}	0.39	0.39
G_{msat}	103 mS/mm	180 mS/mm
V_{tlin}	0.55 V	0.6 V
R_{ext}	500 W - mm	500 W - mm
I_{Dsat}	400 µA/mm	400 µA/mm

CAPs

MOS Cap	1.50 fF/µm^2
MIM Cap	0.70 fF/µm^2

Diodes

SBD (Schottky)	Vf = 300 mV@100 µA
PIN	Vf = 790 mV@100 µA
VARactor	1.4 fF/µm@ 0 V Vf = 810 mV@100 µA
ESD	2000 V HBM

Spiral Inductors* (examples)

4Turn:	Q(2 GHz) = 7.5, 4.2 nH
6Turn:	Q(1 GHz) = 5.8, 9.8 nH
8Turn:	Q(1 GHz) = 5.2, 16.6 nH

Figure 3.1 IBM SiGe BiCMOS process 5HP specifications (From [32].).

Circuit design of the ADCs in this chapter was performed using Tanner Tools SEdit. Unfortunately, the circuit simulator provided by Tanner was not compatible with the circuit element models provided by IBM and, therefore, Silvaco SmartSpice was used for simulation of the ADC circuits. Unfortunately, IBM relies solely on the Cadence design environment for design and simulation of their SiGe processes. Therefore, the format of the Cadence models and model parameters provided by IBM for the 5HP SiGe process were significantly modified to be compatible with the format required by the SmartSpice simulator. Details of the modifications are not published in this dissertation due to IBM proprietary restrictions.

B. THREE-CHANNEL RSNS ANALOG-TO-DIGITAL CONVERTER

The ADC presented in this section is a unipolar, six-bit, three-channel RSNS folding ADC. The moduli are identical to [30] with $m_1 = 3$, $m_2 = 4$, and $m_3 = 5$. Using these moduli, the number of unique RSNS vector residue values, which ultimately determine the dynamic range of the RSNS ADC, is 43. The computation of the number of unique RSNS vector residue values for a given set of moduli is the subject of the Chapter IV. The 43 integer values can be represented with 6 bits with some unused combinations. The least significant bit (LSB) voltage is the smallest input voltage change that causes an output RSNS residue transition. In this design and in [30], the LSB is chosen to be 12 mV. Therefore, the dynamic range of the ADC in volts is

$$(43)(0.012 \text{ mV}) = 0.516 \text{ V}. \tag{3.1}$$

Using notation similar to [30], the folding period for each channel is

$$T_i = 2qNm_i \text{ V}, \tag{3.2}$$

where q is the size of the LSB in volts, N is the number of channels, and the subscript i is the channel number. Thus, the folding periods for each channel are

$$\begin{aligned} T_1 &= 216 \text{ mV}, \\ T_2 &= 288 \text{ mV}, \\ T_3 &= 360 \text{ mV}. \end{aligned} \tag{3.3}$$

Figure 3.2 shows the schematic of the three-channel RSNS folding ADC. The structure is virtually identical to the design in [30]. The labels in the figure are as follows: *sij* are the output bits in thermometer code, *CVij* are the comparator threshold levels, and *compij* are the comparator output, where the index *i* is the channel number and the index *j* is the bit number ($j = 0$ indicates the least significant bit). The three folding amplifiers in the schematic fold the input signal multiple times with a folding period equal to the values in (3.3).

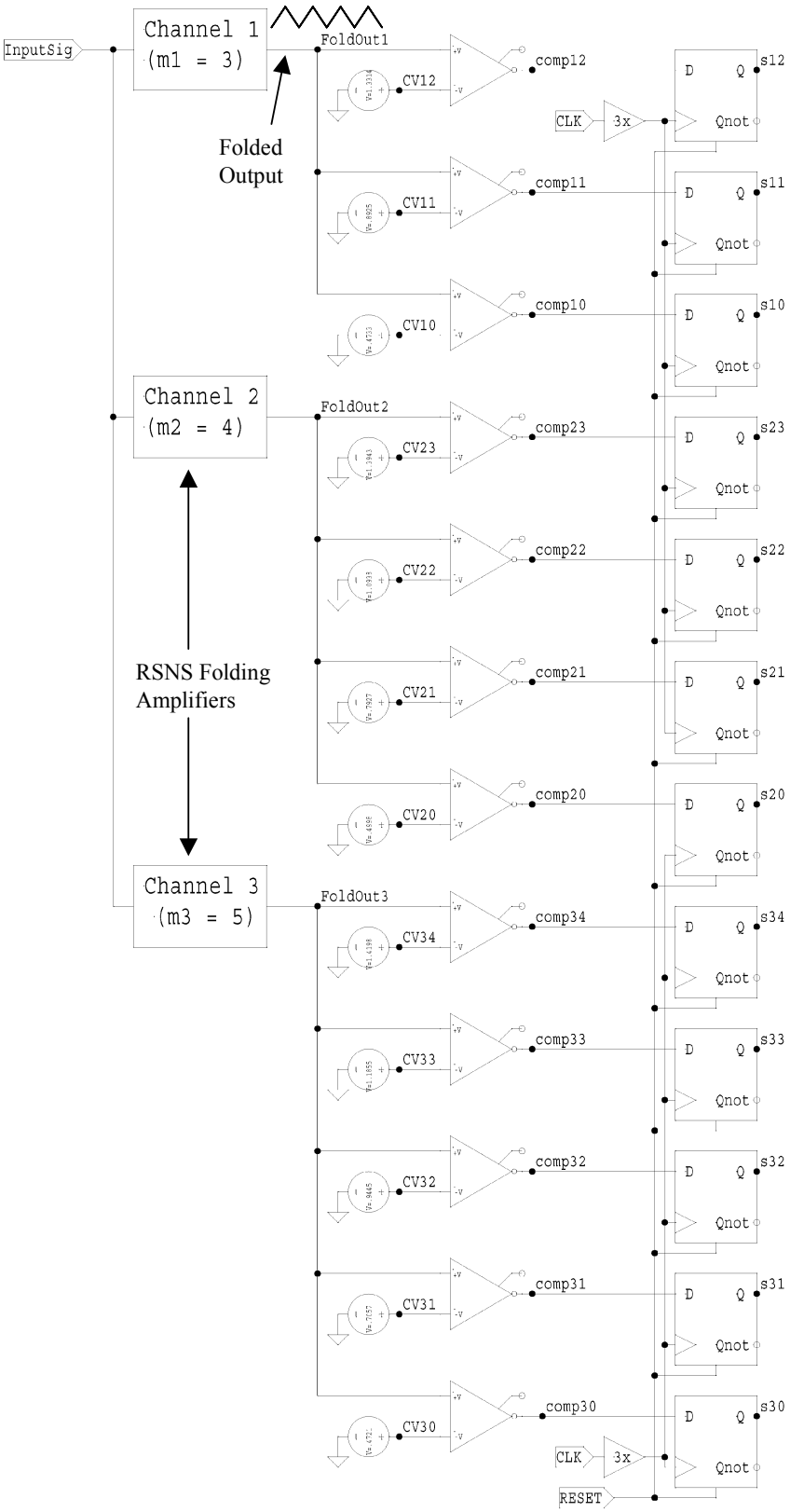

Figure 3.2 Three-channel RSNS folding ADC.

1. Folded Waveform Generation

The folding amplifiers for each channel are composed of several identical, but independent, folding stages and are shown in Figure 3.3, Figure 3.4, and Figure 3.5.

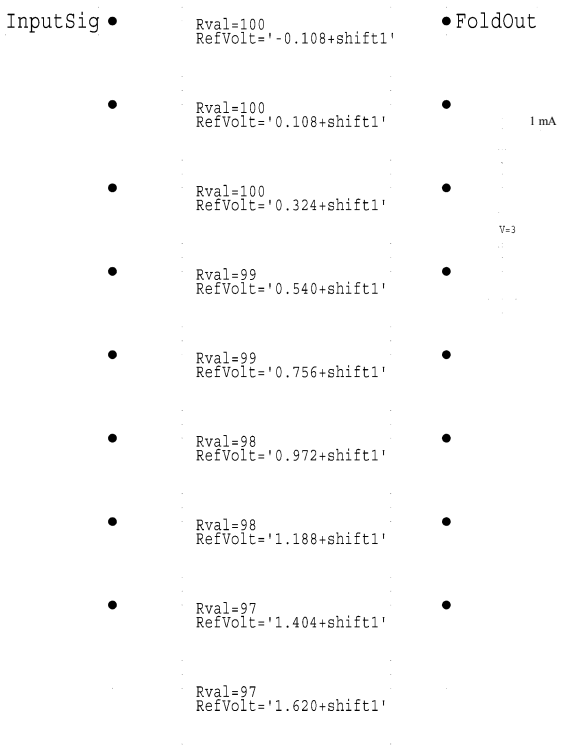

Figure 3.3 Folded waveform generation for the first channel ($m_1 = 3$).

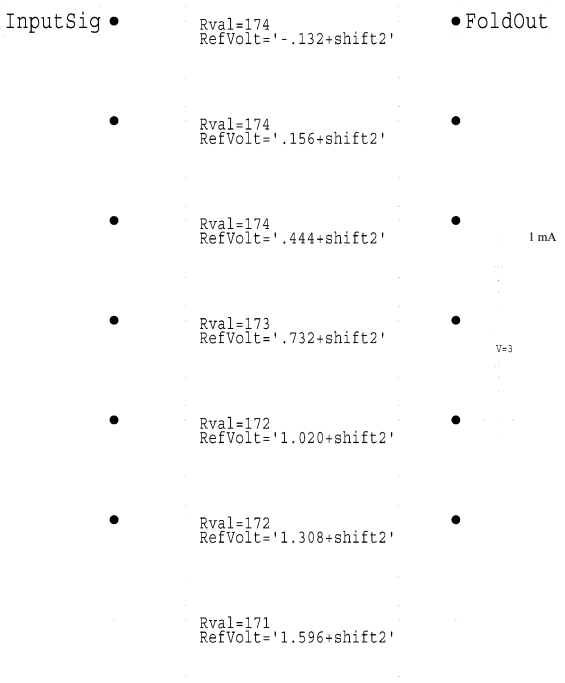

Figure 3.4 Folded waveform generation for the second channel ($m_2 = 4$).

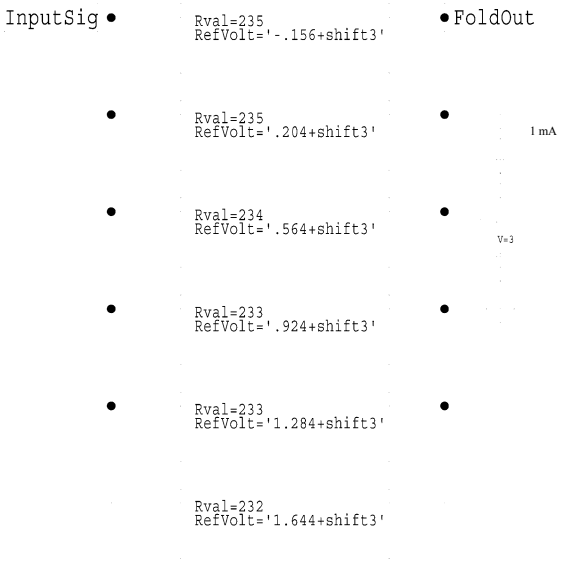

Figure 3.5 Folded waveform generation for the third channel ($m_3 = 5$).

The resistor value *Rval* controls the width of the fold; the voltage *RefVolt* determines the position of the peak of the fold. The voltage increments *shift1*, *shift2*, and *shift3* allow the folded waveforms for each channel to be shifted right or left by multiples of the LSB in order to correctly align the folded waveforms. The folded waveforms need

to be accurately aligned to form the RSNS and also to position the dynamic range such that the code for the first RSNS vector in the longest sequence of unique RSNS vectors occurs at the output of the ADC when the input is at zero volts (or anywhere else desired). Computation of the shift values will be discussed later in this section. The individual folding stage from [30] is shown in Figure 3.6.

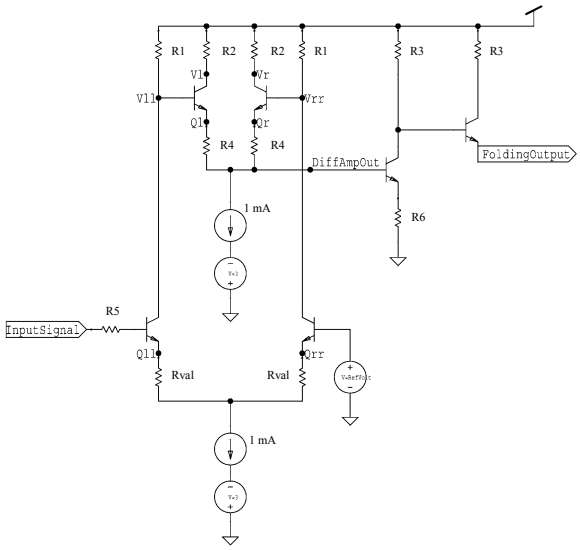

Figure 3.6 Single folding stage circuit.

The resistor values common to all folding amplifier stages are provided in Figure 3.7.

Resistor Label	Value (ohms)
R1	2000
R2	500
R3	900
R4	5
R5	300
R6	400
Rval	see channel tables

Figure 3.7 Folding amplifier common resistor values.

The unique values *Rval* and *RefVolt* for each individual stage in each folding amplifier are provided in Figure 3.8, Figure 3.9, and Figure 3.10. Notice that, for each chan-

channel, the difference in *RefVolt* between successive folds is exactly the value given in (3.3). Notice also that *Rval* varies slightly between folds. This compensates for a slight rise in the valley of each fold as more folds are added. For most applications, an average value for *Rval* can be used.

Channel 1 ($m_1 = 3$)

Fold	Rval (ohms)	Reference Voltage *RefVolt* (V)
1	100	-0.108+shift1
2	100	0.108+shift1
3	100	0.324+shift1
4	99	0.540+shift1
5	99	0.756+shift1
6	98	0.972+shift1
7	98	1.188+shift1
8	97	1.404+shift1
9	97	1.620+shift1

Figure 3.8 Channel one folding amplifier resistor and reference voltage values.

Channel 2 ($m_2 = 4$)

Fold	Rval (ohms)	Reference Voltage *RefVolt* (V)
1	174	-0.132+shift2
2	174	0.156+shift2
3	174	0.444+shift2
4	173	0.732+shift2
5	173	1.020+shift2
6	172	1.308+shift2
7	171	1.596+shift2

Figure 3.9 Channel two folding amplifier resistor and reference voltage values.

Channel 3 ($m_3 = 5$)

Fold	Rval (ohms)	Reference Voltage *RefVolt* (V)
1	235	-0.156+shift3
2	235	0.204+shift3
3	234	0.564+shift3
4	233	0.924+shift3
5	233	1.284+shift3
6	232	1.644+shift3

Figure 3.10 Channel three folding amplifier resistor and reference voltage values.

Each channel must fold the input signal over the entire dynamic range. From [30], the minimum number of folds (folding stages) in each channel is

$$\text{Number of folds} = \left\lceil \frac{\hat{M}}{2Nm_i} \right\rceil, \tag{3.4}$$

where $\hat{M}$ is longest sequence of unique RSNS vectors (RSNS dynamic range). The number of folding stages shown in Figure 3.3, Figure 3.4, and Figure 3.5 are more than the minimum number of stages computed using (3.4). This is because this ADC implementation in this chapter is used to test various aspects of the RSNS, including the capability to move the dynamic range to any point by varying value of the parameters *shift1*, *shift2*, and *shift3*. Therefore, the number of folding stages in each channel is much larger than minimum to accommodate the dynamic range shift during testing. The folding waveforms at the output of each folding amplifier are amplitude analyzed using a parallel configuration of latched comparators as shown in Figure 3.2.

2. Latched Comparator Design

In most ADC designs, the comparator is one of the two most critical elements in terms of affecting the speed and power consumption of the circuit. Extra attention must be paid to maximizing the clock rate while maintaining low-power operation. A comparator consists of an amplifier and a latch. A key feature of the RSNS folding ADC design is that the number of comparators is minimized. With the RSNS, the number of comparators is the sum of the channel moduli. In this case, the number of comparators is twelve as shown in Figure 3.2. Alternative high-speed ADC designs, such as a flash ADC, require as many as 63 comparators for a six-bit ADC (32 comparators using interpolation), and 14 comparators for a conventional folding ADC. Thus, the die area savings and power reduction are significant using RSNS techniques. The comparator for this ADC is shown in Figure 3.11. The resistor values are the same as for the folding circuits and are listed in Figure 3.7

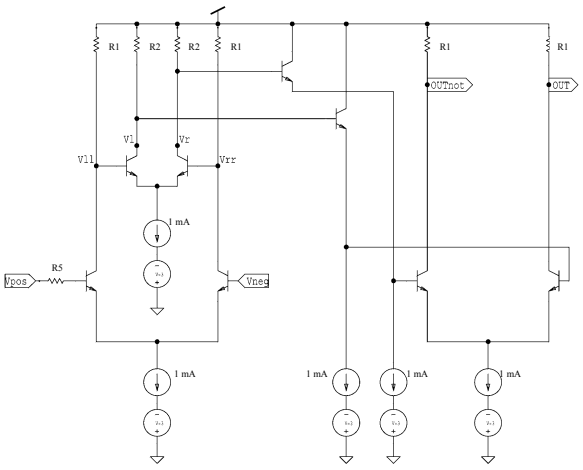

Figure 3.11 Comparator design.

The comparator consists of nested differential amplifiers buffered to a second differential amplifier. The nested differential amplifier evaluates the two comparator inputs and the second differential amplifier amplifies the comparator output so that the voltage swing is compatible with CMOS circuits. The labels *OUT* and *OUTnot* in the schematic above correspond to the non-inverted (*compij*) and inverted outputs of the comparator symbols in Figure 3.2. The comparator latch latches the amplified difference and is formed from a D-type flip-flop shown in Figure 3.12 where the latched output is *Q* and the inverted output is *Qnot*.

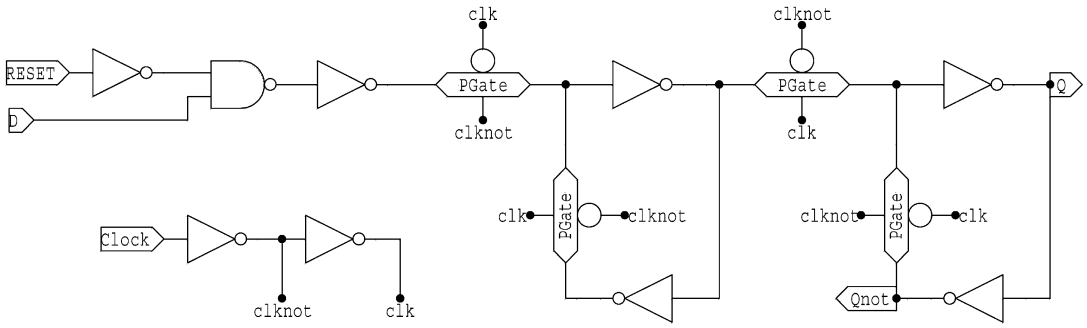

Figure 3.12 Comparator latch using D-type flip-flop.

The comparator input is labeled *D* and is connected to the non-inverted comparator output *compij* as shown in Figure 3.2. The comparator used in this design is not the fastest or most power efficient available but it provides adequate speed and serves to il-

lustrate the benefits of the RSNS folding ADC. Notice that although the circuits for the latched comparator and the single folding amplifier stage are similar, the comparator has approximately twice the number of circuit elements. This observation will be useful when considering *N*-channel RSNS ADC designs in the next section.

The number of comparators associated with each channel of the RSNS ADC is equal to the modulus of the channel. The voltage reference level for each comparator is set such that the voltage between the comparator threshold level intersections with the folded waveform are exactly three LSBs apart. Figure 3.13 shows a plot of the folded waveform, comparator threshold levels, and the comparator output for the first channel. Note that in each channel, a comparator threshold is crossed every Nq. That is, only one comparator threshold is crossed at a time within the system and alternates between the channels every q volts.

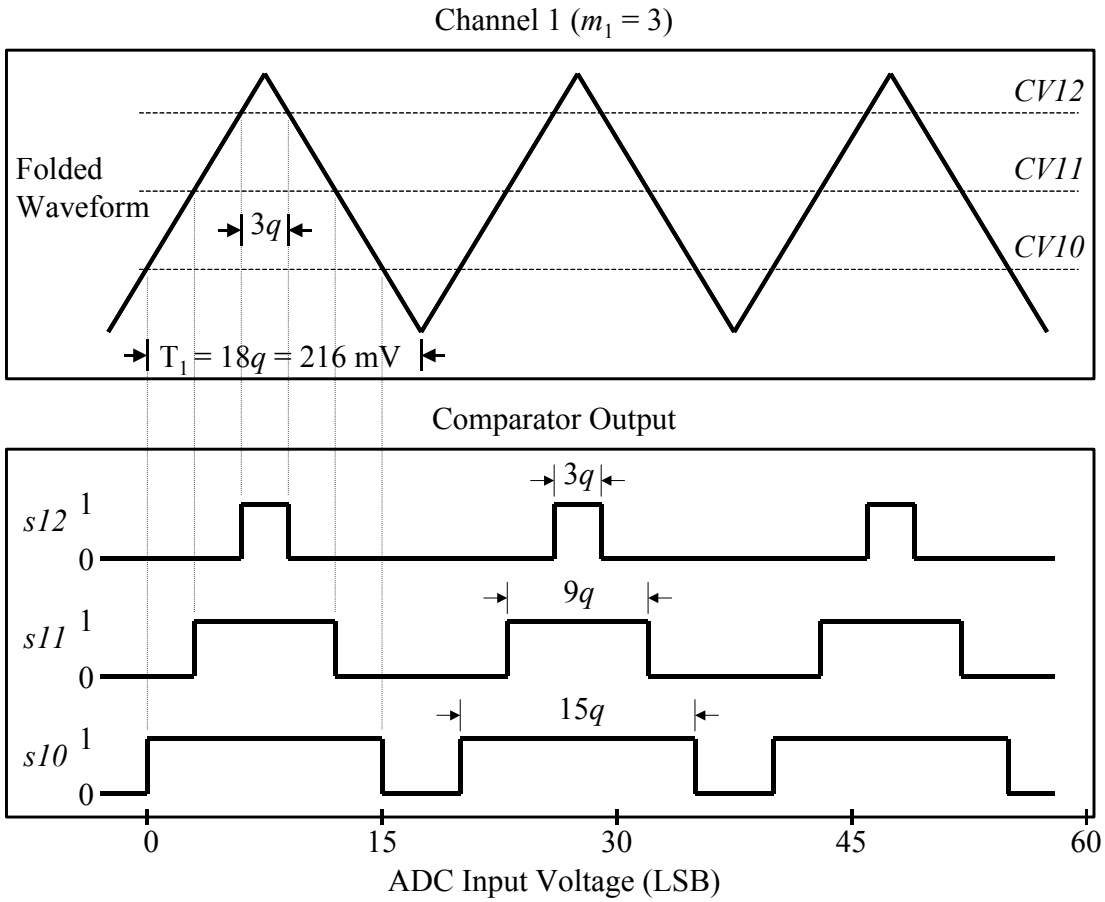

Figure 3.13 Graph of folded waveform and comparator output (After [30].).

Similar plots are provided in [30] for all three channels. These results can be extended to the *N*-channel ADC where the folded waveform and comparator threshold intersections are *N* LSBs apart.

The output from the bank of comparators is usually referred to as a thermometer code due to the particular form of the output. Figure 3.14 shows the equivalent decimal value, binary value, and thermometer code for a three-bit binary number.

Decimal Value	Binary Value	Thermometer Code
0	000	0000000
1	001	0000001
2	010	0000011
3	011	0000111
4	100	0001111
5	101	0011111
6	110	0111111
7	111	1111111

Figure 3.14 Thermometer code with binary and decimal equivalents.

Summing the thermometer code output of the three comparators in the first channel yields the characteristic RSNS stair-step graph in Figure 3.15. A similar graph can be formed for the second and third channels.

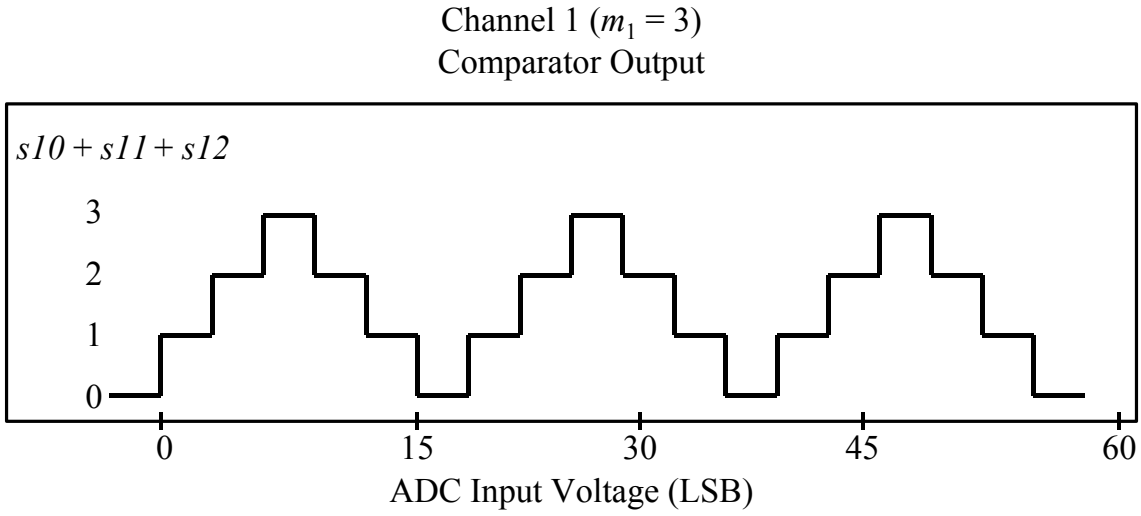

Figure 3.15 Summed comparator stair-step output.

One method of computing the comparator thresholds is to perform a DC analysis simulation in SPICE for each of the folding amplifiers in Figure 3.3, Figure 3.4, and Figure 3.5. In each of the DC analysis simulations, the SPICE code lines shown in Figure 3.16 (or something similar depending on the version of SPICE used) located the correct comparator threshold values for the first folding channel as well as the peak and valley voltages. In the SPICE code provided, *q* is the LSB and *RefVolt* is any fold peak voltage from the third column of Figure 3.8. This concept can be extended to find the comparator values for the second and third channels as well.

```
.measure dc FoldPeak find v(FoldOut) when InputSig='RefVolt'
.measure dc CV12     find v(FoldOut) when InputSig='RefVolt +(3*q*0.5)'
.measure dc CV11     find v(FoldOut) when InputSig='RefVolt +(3*q*1.5)'
.measure dc CV10     find v(FoldOut) when InputSig='RefVolt +(3*q*2.5)'
.measure dc FoldValy find v(FoldOut) when InputSig='RefVolt +(3*q*3.0)'
```

Figure 3.16 SPICE code to compute comparator thresholds.

All comparator threshold values for the three-channel RSNS folding ADC shown in Figure 3.2 were computed using this method and are provided in Figure 3.17.

Comparator Input	Threshold Voltage (V)
CV10	0.4733
CV11	0.8925
CV12	1.3314
CV20	0.4998
CV21	0.7927
CV22	1.0938
CV23	1.3943
CV30	0.4721
CV31	0.7057
CV32	0.9445
CV33	1.1855
CV34	1.4198

Figure 3.17 Comparator threshold values for three-channel RSNS folding ADC.

C. FOUR-CHANNEL RSNS ANALOG-TO-DIGITAL CONVERTER

The DIS requires eight-bit *I* and *Q* digital inputs, which must be converted from analog signals by means of ADCs. To realize an eight-bit binary output, the longest sequence of unique RSNS residue vectors ($\hat{M}$) must be greater than or equal to 256. Thus, the *N* moduli for the *N*-channel eight-bit ADC must produce this unambiguous RSNS range. Only three-modulus and four-modulus systems were considered for the eight-bit ADC since pair-wise relatively prime RSNS moduli sets for $N \geq 5$ produce an $\hat{M}$ that is much greater than 256. Methods of computing $\hat{M}$ for any moduli set will be shown in the next chapter. Figure 3.18 shows a small sample of the hundreds of possible moduli sets that produce an $\hat{M} \cong 256$. The *Moduli Sum* column indicates the total number of comparators required.

Moduli Sum	m_1	m_2	m_3	m_4	$\hat{M}$
32	9	10	13		276
34	10	11	13		264
34	10	11	13		264
35	7	11	17		267
35	7	11	17		267
35	7	13	15		312
36	9	11	16		258
36	7	10	19		261
27	3	5	8	11	265
27	4	5	7	11	287
28	3	5	7	13	312
29	3	5	8	13	285
29	4	5	7	13	286
29	3	7	8	11	287
29	5	6	7	11	360
29	5	7	8	9	407

Figure 3.18 Three-modulus and four-modulus RSNS moduli with $\hat{M} \cong 256$.

Only the RSNS moduli combinations with the lowest sum were included since the moduli set with the lowest sum minimizes the number of comparators in the ADC. Moreover, the number of folding amplifiers required is equal to the number of moduli, and the approximate number of individual folding stages for each amplifier is computed using (3.4). Figure 3.19 shows the number of comparators (moduli sum) and individual folding stages for each of the moduli combinations in Figure 3.18. Since the output of the ADC is limited to eight bits, the RSNS dynamic range was truncated to 256 for all the moduli sets. Therefore, the computation of the values in the following table assumed that $\hat{M} = 256$. The last column in the figure provides a computation of *comparator equivalents* assuming that the folding stages are approximately half the size of the comparators.

m_1	m_2	m_3	m_4	Folding Amplifiers	m_1 stages	m_2 stages	m_3 stages	m_4 stages	Folding Stages	Comparators	Comparator Equivalents
9	10	13		3	5	5	4		14	32	39
10	11	13		3	5	4	4		13	34	40.5
10	11	13		3	5	4	4		13	34	40.5
7	11	17		3	7	4	3		14	35	42
7	11	17		3	7	4	3		14	35	42
7	13	15		3	7	4	3		14	35	42
9	11	16		3	5	4	3		12	36	42
7	10	19		3	7	5	3		15	36	43.5
3	5	8	11	4	11	7	4	3	25	27	39.5
4	5	7	11	4	8	7	5	3	23	27	38.5
3	5	7	13	4	11	7	5	3	26	28	41
3	5	8	13	4	11	7	4	3	25	29	41.5
4	5	7	13	4	8	7	5	3	23	29	40.5
3	7	8	11	4	11	5	4	3	23	29	40.5
5	6	7	11	4	7	6	5	3	21	29	39.5
5	7	8	9	4	7	5	4	4	20	29	39

Figure 3.19 Computation of comparator equivalents for the moduli in Figure 3.18.

For the four-channel ADC design, a moduli combination with a low number of comparator equivalents was chosen, which is the four-modulus RSNS with $m_i = [3 \quad 5 \quad 8 \quad 11]$. Using these moduli, the number of comparators required is 27, the number of folding amplifiers required is 4, and the total number of individual folding stages is 25. In retrospect, $m_i = [5 \quad 7 \quad 8 \quad 9]$ may have been a better choice since the

comparator equivalents are low and the number of comparator levels per channel is limited to nine instead of eleven. The reduction in comparator levels relaxes the comparator tolerances and allows the ADC to operate at higher frequencies. Moreover, although the number of comparator equivalents is approximately the same for the three-modulus and four-modulus RSNS for this particular $\hat{M}$, converting the RSNS residues to binary is much simpler for the four-modulus case than the three-modulus case. RSNS-to-binary conversion is covered in detail in Chapter V.

Using the same 3.3-V IBM SiGe fabrication process described in the first section, the LSB was chosen to be 6 mV, which yielded the ADC dynamic range

$$(256)(0.006\text{ mV}) = 1.536\text{ V}. \tag{3.5}$$

The schematic for the eight-bit, four-channel RSNS folding ADC is shown in Figure 3.20. The four folding amplifiers and comparators are connected in exactly the same way as the three-channel ADC in the previous section. For simplicity, only the details of the first channel are shown for the four-channel ADC. Figure 3.21 provides the comparator circuit block, Figure 3.22 shows the clock distribution block, and Figure 3.23 shows the folding amplifier block.

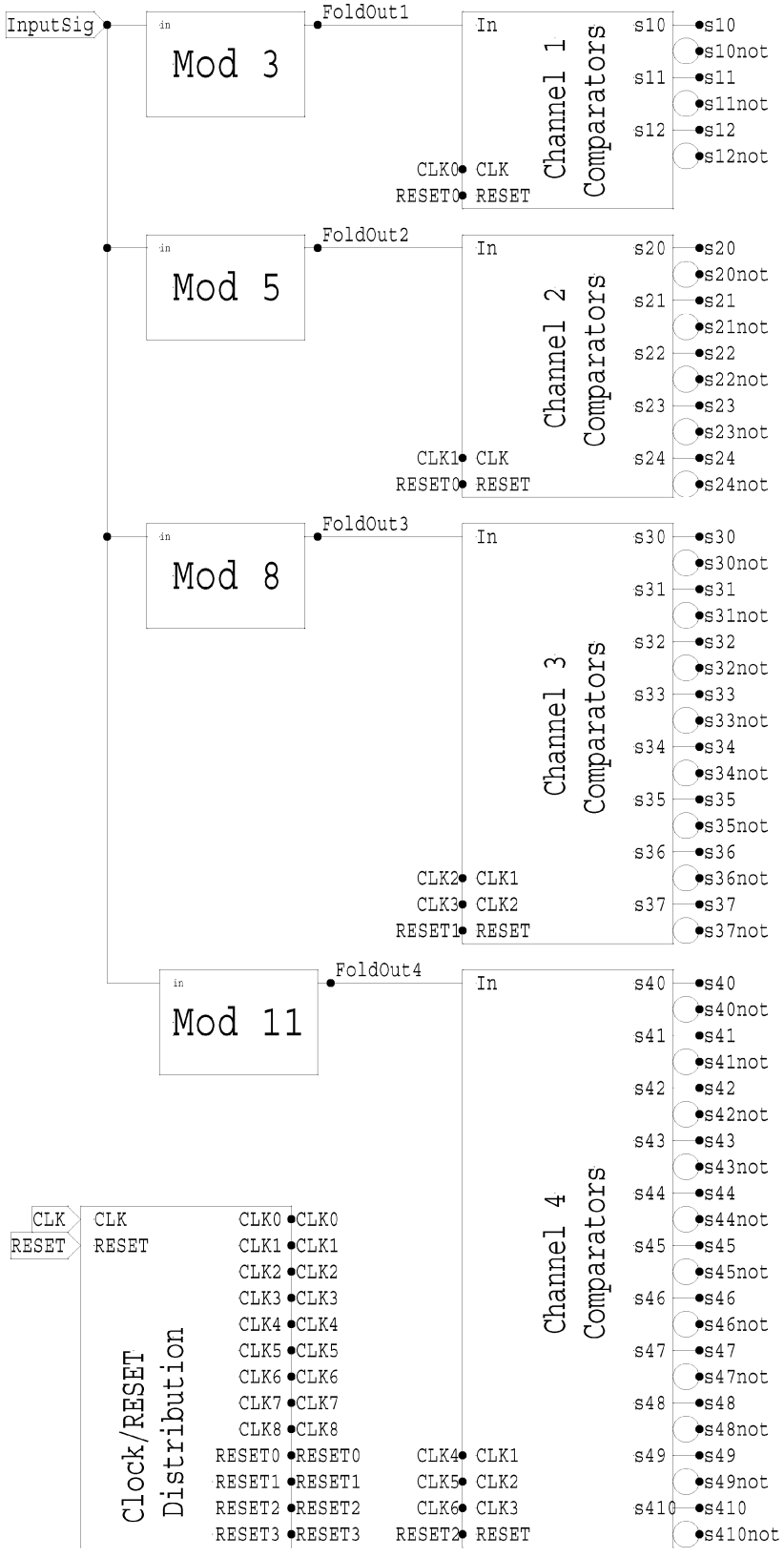

Figure 3.20 Four-channel RSNS folding ADC.

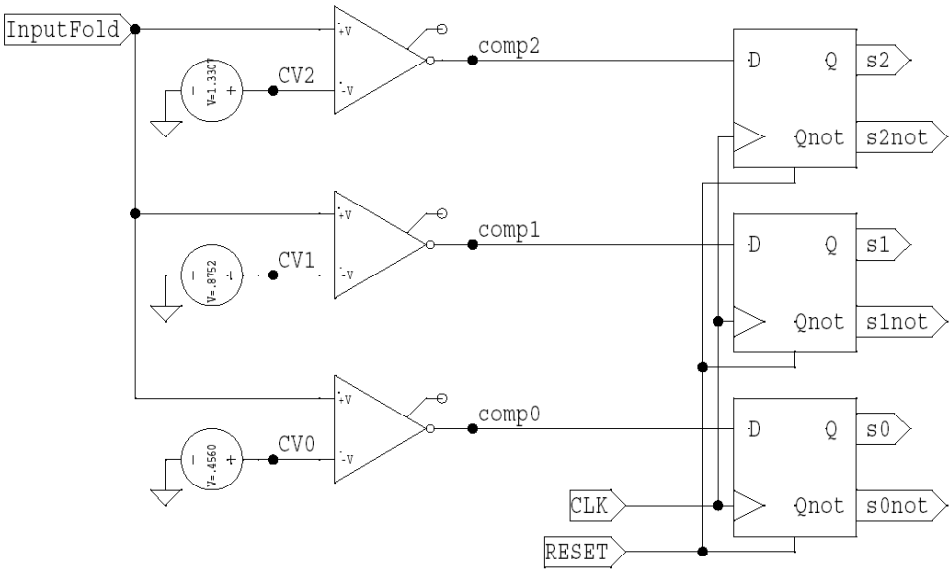

Figure 3.21 Channel one comparator block in four-channel RSNS ADC.

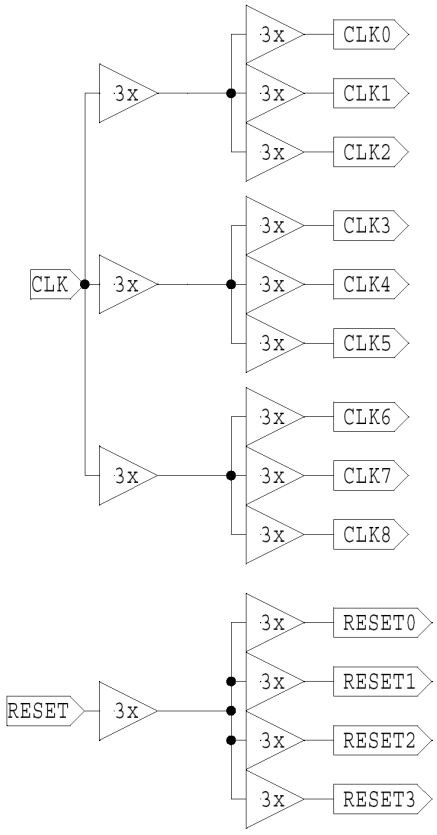

Figure 3.22 Clock distribution block in four-channel RSNS ADC.

Figure 3.23 Folding amplifier block for $m_i = 3$ for the four-channel RSNS ADC.

The resistor values common to all individual folding amplifiers for the four-channel ADC are the same as the three-channel ADC and are given in Figure 3.7. Figure 3.24 shows the unique values for *Rval* and *RefVolt* for each folding stage in each channel.

Fold	Channel 1 ($m_1 = 3$)		Channel 2 ($m_2 = 5$)		Channel 3 ($m_3 = 8$)		Channel 4 ($m_4 = 11$)	
	Rval (ohms)	Reference Voltage (V)	Rval (ohms)	Reference Voltage (V)	Rval (ohms)	Reference Voltage (V)	Rval (ohms)	Reference Voltage (V)
1	40	-0.072+shift1	130	-0.120+shift2	250	-0.192+shift3	350	-0.264+shift4
2	40	0.072+shift1	130	0.120+shift2	250	0.192+shift3	350	0.264+shift4
3	40	0.216+shift1	130	0.360+shift2	250	0.570+shift3	350	0.792+shift4
4	40	0.360+shift1	130	0.600+shift2	250	0.960+shift3	350	1.320+shift4
5	40	0.504+shift1	130	0.840+shift2	250	1.344+shift3		
6	40	0.648+shift1	130	1.080+shift2				
7	40	0.792+shift1	130	1.320+shift2				
8	40	0.936+shift1	130	1.560+shift2				
9	40	1.080+shift1						
10	40	1.224+shift1						
11	40	1.368+shift1						
12	40	1.512+shift1						

Figure 3.24 Reference voltages for individual folding stages in each channel.

The eight-bit RSNS ADC illustrates a problem that was not as apparent in the six-bit ADC. That is, the magnitude of the dynamic range voltage is dependent on the position of $\hat{M}$ in the fundamental period of the RSNS. For the six-bit ADC, $\hat{M}$ started at vector 79 and stopped at vector 122, which is equivalent to vectors 79 to 121, inclusive. Multiplying by the LSB from the previous section (12 mV) results in a dynamic range starting at 0.948 V and ending at 1.464 V. These values are well within the 3.3-V range of the 5HP fabrication process. However, for the eight-bit ADC with $\hat{M}$ truncated to 256, the start vector was at 2616 and the stop vector was at 2872. Multiplying by the eight-bit ADC LSB (6 mV) results in a dynamic range starting at 15.696 V and ending at 17.232 V, which is clearly outside of the voltage limits for the fabrication process. There are two methods to alleviate this problem. The first solution is to choose a set of moduli that has an $\hat{M}$ that starts at a position in the RSNS fundamental period at or near zero. For instance, the moduli set $m_i = [9 \quad 10 \quad 13]$ from Figure 3.19 has an $\hat{M}$ that starts at zero. However, in the general case there is no guarantee that a practical moduli set with an $\hat{M}$ and a start position compatible with a particular voltage range even exists. A second technique involves shifting each channel individually so that the start of the dynamic range is within the voltage range of the fabrication process. This technique can ensure that the code for the first $\hat{M}$ vector occurs at the output of the ADC when the input is

zero volts, which is particularly desirable for a unipolar ADC. The procedure is simply to shift each channel left by an LSB multiple equal to the start position of $\hat{M}$ (mod $2Nm_i$). Note that this shift can be positive or negative. This is in addition to the shift already required to form the RSNS. As an example, the *Refvolt* shift for each channel in the eight-bit ADC described above with an $\hat{M}$ start position of $h = 2616$ is

$$\begin{aligned} shift_1 &= -(0 + 2616 \pmod{2Nm_1})\text{LSB} = -(0+0)(0.006) = 0 \text{ mV}, \\ shift_2 &= -(1 + 2616 \pmod{2Nm_2})\text{LSB} = -(1-24)(0.006) = 0.138 \text{ mV}, \\ shift_3 &= -(2 + 2616 \pmod{2Nm_3})\text{LSB} = -(2-8)(0.006) = 0.036 \text{ mV}, \\ shift_4 &= -(3 + 2616 \pmod{2Nm_4})\text{LSB} = -(3-24)(0.006) = 0.126 \text{ mV}. \end{aligned} \tag{3.6}$$

Of course, shifting the folding waveform necessitates the inclusion of an additional individual folding stage circuit in some of the folding amplifiers, because the folding amplifiers must fold the input signal plus any added shift voltage.

Finally, to complete the design description, the comparator threshold voltages were computed using the technique from Figure 3.16 and are provided in Figure 3.25.

Comparator Input	Voltage Level (V)
CV10	0.456
CV11	0.8752
CV12	1.3307
CV20	0.4595
CV21	0.6918
CV22	0.9331
CV23	1.179
CV24	1.4188
CV30	0.4148
CV31	0.599
CV32	0.7077
CV33	0.8584
CV34	1.0104
CV35	1.163
CV36	1.3153
CV37	1.4454
CV40	0.3507
CV41	0.4482
CV42	0.5598
CV43	0.6728
CV44	0.7868
CV45	0.9015
CV46	1.0166
CV47	1.1319
CV48	1.247
CV49	1.361
CV410	1.4491

Figure 3.25 Comparator threshold values for four-channel RSNS folding ADC.

D. SIMULATION RESULTS

All simulations were performed using Silvaco SmartSpice using the IBM SiGe 5HP fabrication process models and parameters described previously in this chapter. Post-simulation processing and plotting was carried out in MATLAB. Each component of the ADC was simulated individually to ensure correct operation. For the simulation of the entire ADC circuit, a voltage ramp was presented as an input to the ADC and the RSNS thermometer code was observed at the comparator output.

1. Three-Channel RSNS ADC

For the three-bit ADC ramp-input simulation, the clock rate for the comparators was fixed at 1 GHz and voltage ramp rate was set to 0.005 volts per nanosecond. Since the LSB was chosen to be 0.012 V, there were approximately two clock periods within each LSB. Figure 3.26 shows the output of the folding amplifiers and the comparator threshold levels, which coincide with the results in [30]. Figure 3.27 shows simulation results for the six-bit ADC. The top plot shows the input ramp and the bottom three plots show the summed comparator output.

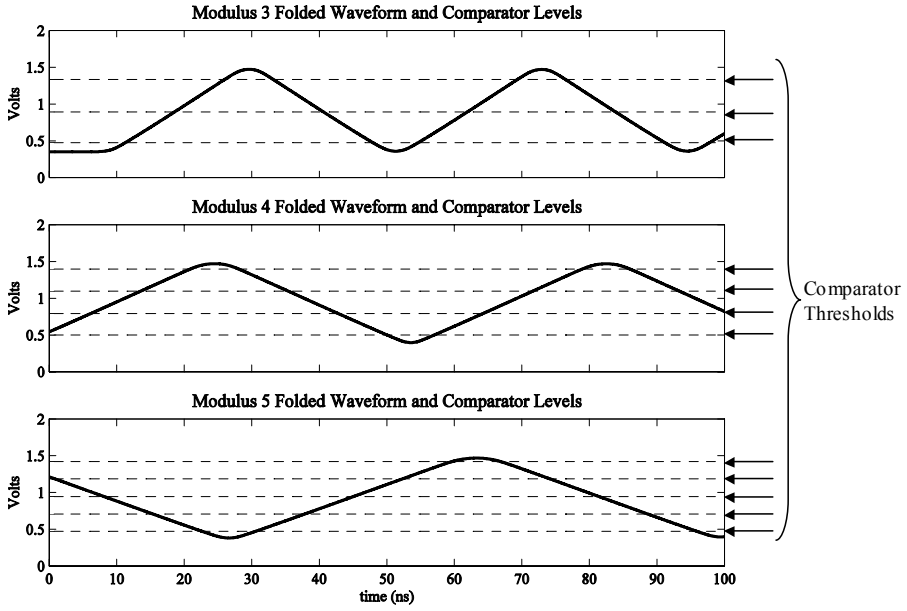

Figure 3.26 Folded waveforms and comparator levels for six-bit ADC.

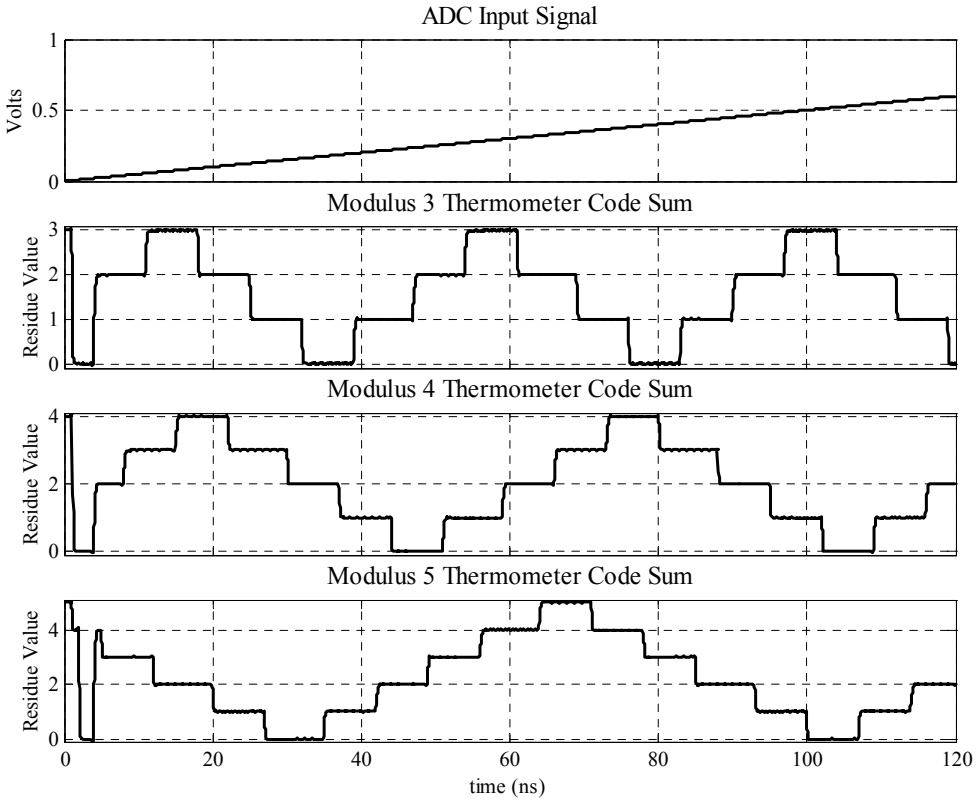

Figure 3.27 ADC simulation showing input signal and folded output.

The thermometer code comparator output was converted to binary using an RSNS-to-binary conversion algorithm discussed in detail in Chapter V. The three-channel ADC binary output converted to decimal is shown in Figure 3.28.

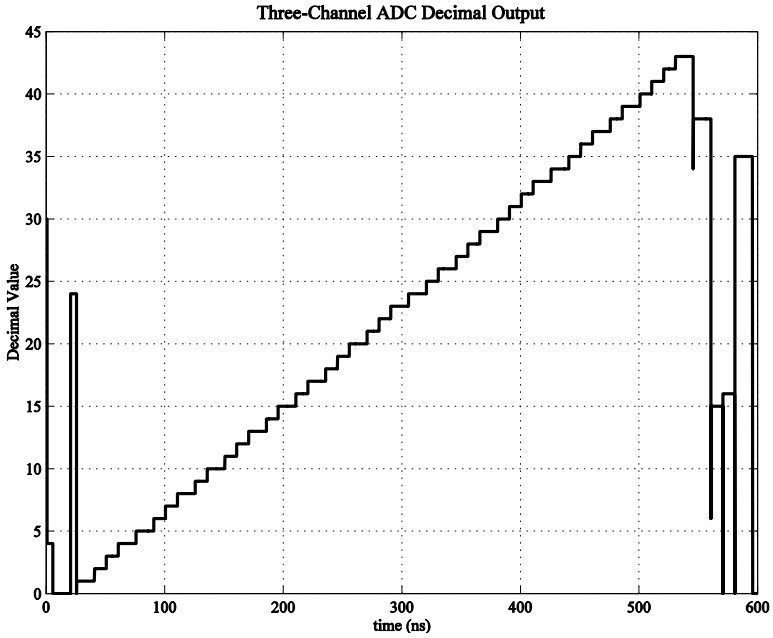

Figure 3.28 ADC simulation showing stair-step decimal output.

The *differential nonlinearity* (DNL) is the maximum deviation in the output step size from the ideal value of one LSB, while the *integral nonlinearity* (INL) is the maximum deviation of the output stair step from a straight line passed through its endpoints [28]. The INL and DNL for the three-channel ADC are shown in Figure 3.29. An ADC is considered to have good linearity if it has an INL and DNL of less than one half of a LSB [28]. Figure 3.28 shows that the three-channel RSNS folding ADC is well within that standard for both INL and DNL.

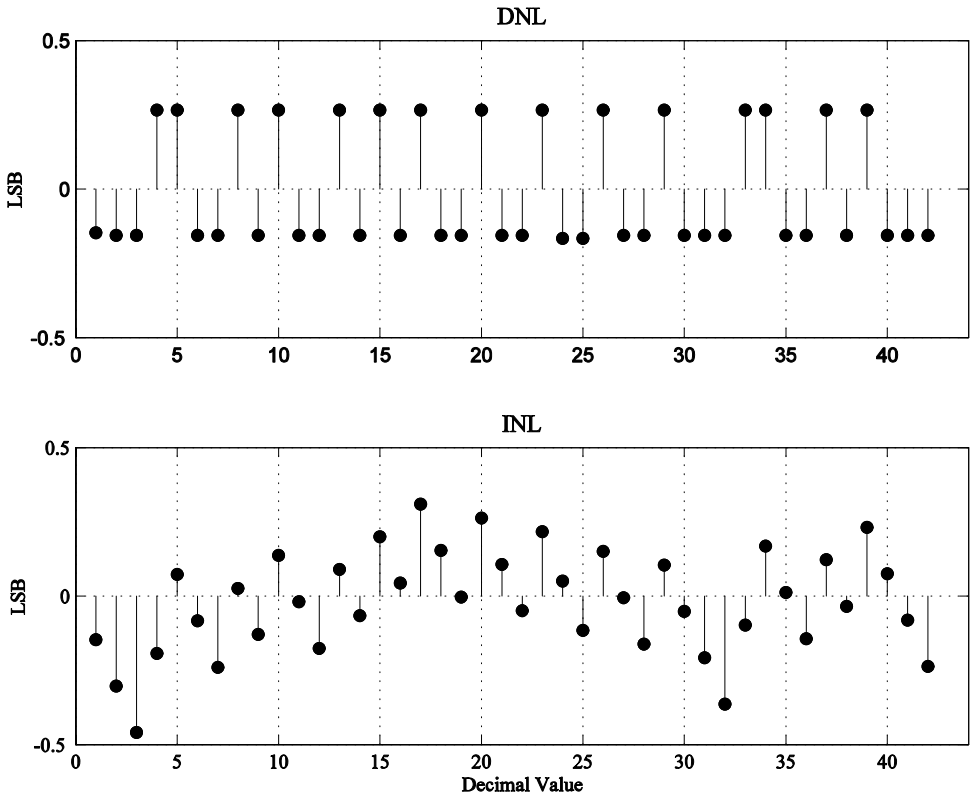

Figure 3.29 INL and DNL for the six-bit ADC.

2. Four-Channel RSNS ADC

For the eight-bit ADC ramp-input simulation, the clock rate for the comparators was fixed at 1 GHz and ramp rate was set to 0.0025 volts per nanosecond. Since the LSB was chosen to be 0.006 V, there were approximately 2 clock periods within each LSB. Figure 3.30 shows simulation results for the eight-bit ADC. The top plot shows the input ramp and the bottom four plots show the summed comparator output.

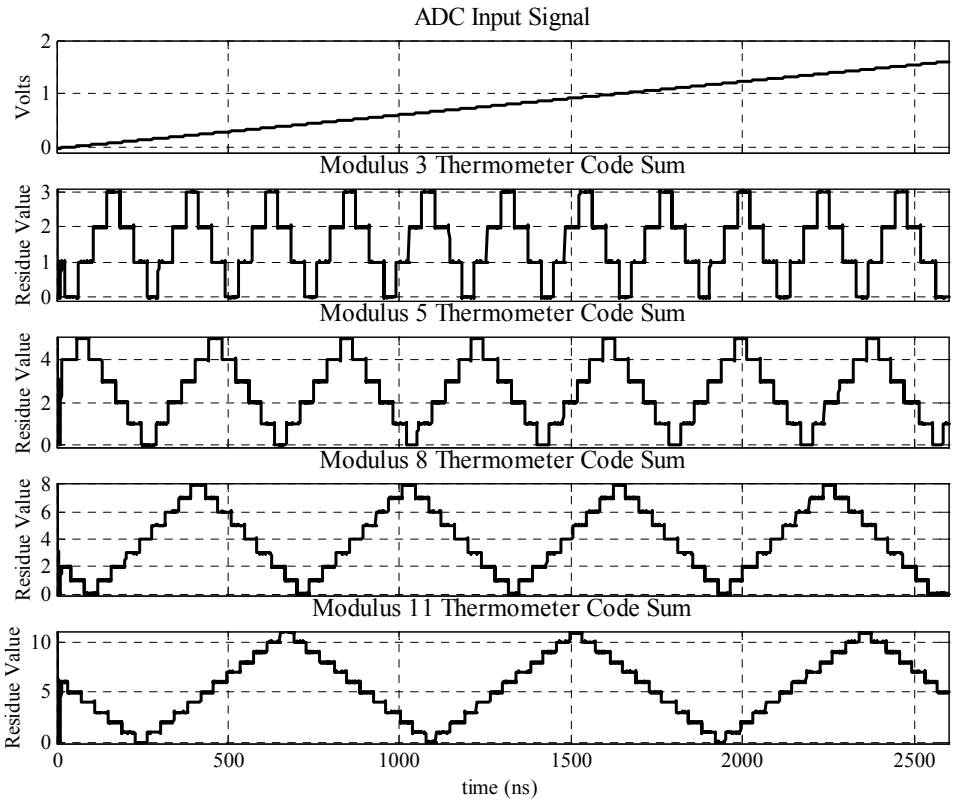

Figure 3.30 ADC simulation showing input signal and folded output.

The thermometer code comparator output was converted to binary using an RSNS-to-binary conversion algorithm discussed in detail in Chapter V. The INL and DNL are shown in Figure 3.31. The INL and DNL in the figure that are greater than half an LSB are due to the fact that the comparators chosen for this implementation probably do not have the resolution to support a 6-mV LSB.

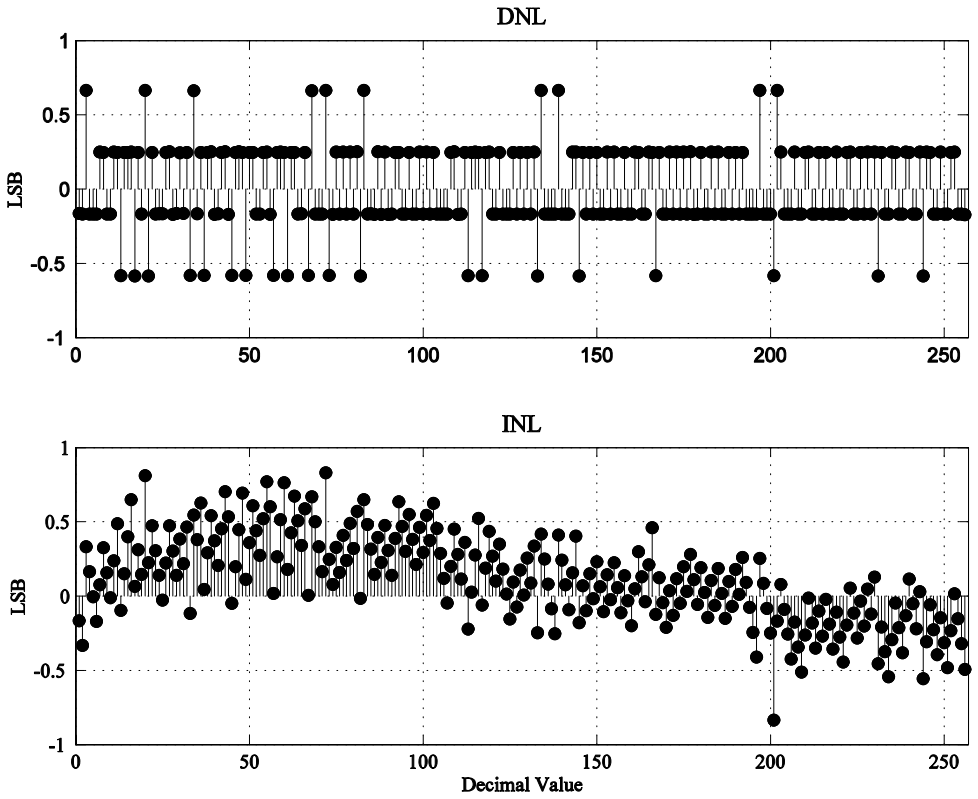

Figure 3.31 INL and DNL for the eight-bit ADC.

E. SUMMARY

The ADC is the most fundamental building block in a signal processing system. To realize a single-chip electronic decoy system, the SoC ADC must have a small die footprint and operate at extremely low power. Folding ADC architectures offer the most resolution at high speeds of any existing ADC architecture as well as a reduced die area and power consumption by minimizing the number of comparators and eliminating interpolation circuitry.

The three-channel, six-bit RSNS ADC extended the design described in [30] by modifying the circuit for implementation in a mixed-signal fabrication process.

The lessons learned designing the six-bit ADC led directly to the design procedure for a four-channel, eight-bit RSNS ADC. An eight-bit ADC is necessary for the implementation of the false target EW SoC since the inputs to the DIS described in Chapter

II are 8-bit digital *I* and *Q* signals, and thus require two 8-bit ADCs to convert the analog *I* and *Q* signals to digital. Furthermore, the eight-bit ADC is used to provide a the procedure for designing *N*-channel *k*-bit RSNS ADCs. The simulation results for both RSNS ADCs verified error-free high-speed operation.

The next chapter represents a departure from the EW SoC component-centric dissertation presented up to this point. Chapter IV describes RSNS in detailed mathematical terms and examines the unique properties of the number system that produce the RSNS error-tolerant Gray-code properties. The RSNS analysis enables the design of *N*-channel ADCs as well as the RSNS-to-binary conversion circuits presented in Chapter V.

IV. ROBUST SYMMETRIC NUMBER SYSTEM

This chapter presents the number theory behind the Robust Symmetrical Number System (RSNS) used to design the ADCs in Chapter III. Up to this point, comprehensive analysis on the RSNS was limited to systems with two moduli [33] and a special three-modulus case [30]. This chapter extends the current two-modulus RSNS theory and analysis to provide the closed-form analytic expressions and search algorithm code to efficiently compute the size and location of the largest sequence of unique RSNS vectors ($\hat{M}$) for three-modulus and *N*-modulus RSNS. The results in this chapter enable the implementation of efficient *N*-modulus, *k*-bit ADCs as well as the RSNS-to-binary converters in Chapter V.

Like *residue number systems* (RNS), s*ymmetrical number systems* (SNS) use modulo arithmetic to decompose an integer value into one or more residue integers. Unlike the RNS and other SNS, the RSNS has a structure with built-in redundancy that is specifically designed to eliminate encoding errors in electronic or mechanical systems that quantize analog signals into a digital integer representation. Thus, all published RSNS research as well as this dissertation present the theory of the RSNS in terms of its application to real systems rather than in abstract mathematical terms.

In an RSNS employing *N* relatively prime moduli $\{m_1\ m_1 \cdots m_N\}$, any integer can be converted to an *N*-dimensional vector of RSNS integer residues. Using notation similar to [28], *a* is a real number and $h = \lfloor a \rfloor$ is the greatest integer less than or equal to *a*. The *N*-dimensional RSNS residue vector representing *h* is denoted X_h. An individual residue in the vector is denoted x_{h_i}, where *i* is the vector row index. In the RSNS, when comparing the vectors X_h and X_{h+1}, all elements of the vectors are the same except for the elements in a single row, and the elements in the row that are not the same vary by exactly ± 1. This Gray-code property makes the RSNS particularly useful in folding analog-to-digital converters [30], direction finding interferometer antenna architectures [34], and electro-optic digital antennas [35] since it eliminates encoding errors common in the

previously mentioned systems. In these applications, the analog signal is folded a number of times (preprocessed) before digitization.

An *ambiguity* is a repeated residue or repeated vector of residues. Because of the presence of ambiguities, the set of integers within each symmetrical number system modulus residue sequence do not form a complete residue system by themselves. The ambiguities are resolved by taking into account the combined values from all modulus residue sequences [28]. Thus, in most practical systems, the vector X_h must be decoded to find the integer *h* as an approximation to the real number *a*. Since the RSNS is periodic, the unambiguous decoding of *h* can only be accomplished within a finite range of vectors. This range of vectors, denoted $\hat{M}$, is the largest series of consecutive non-redundant RSNS vectors. For residue number systems, $\hat{M}$ is simply denoted *M* and is the product of the moduli. However, the ambiguities in the RSNS reduce $\hat{M}$ compared to the RNS and other SNS and is very difficult to compute [35]. Recently, formulas for computing the *length* of $\hat{M}$ for a two-modulus RSNS have been found [33]. Computer search algorithms, however, have to be used to find the *position* of $\hat{M}$ in the RSNS fundamental period for the two-modulus case. Because of the large RSNS fundamental periods, this approach becomes cumbersome and slow for systems with many moduli or systems with large moduli.

In this chapter, the two-modulus RSNS results in [33] are extended to determine the length and position of $\hat{M}$ for a three-modulus and, subsequently, *N*-modulus RSNS. The technique in [33] focused on computing the locations of all redundant vectors rather than searching for sequences of non-redundant vectors. That is, it was more efficient to compute the *finite* locations of the redundancies than to search the entire fundamental period for a sequence of non-redundant vectors of *unknown* length. The technique involved identifying basic redundancies for each modulus residue sequence, combining the basic redundancies into four cases, and then solving the cases to produce an analytic formula for the length of the two-modulus $\hat{M}$. This chapter starts by identifying all modulus residue sequence redundancies and then combines the modulus residue sequence redundancies across all moduli to solve for all vector redundancy locations for the three-

modulus RSNS. The end result is an analytical expression for the length and position of $\hat{M}$ for a popular set of moduli $m-1,\ m,\ m+1$, and an efficient $\hat{M}$ search algorithm for the *N*-modulus RSNS.

This chapter is organized into five sections. The first section develops the structure of the three-modulus RSNS and defines the three fundamental modulus residue sequence redundancies in the system. The next section organizes the redundancies and moduli combinations into several distinct cases and provides analytical solutions to the corresponding redundant vector locations. The three-modulus RSNS results are then extended to produce a comprehensive description and solution for the *N*-modulus RSNS. The last two sections develop an expression for the location and length of $\hat{M}$ for the three-modulus RSNS with moduli of the form $m-1,\ m,\ m+1$, introduce a novel geographic RSNS display technique, and produce an efficient search algorithm for the *N*-modulus RSNS.

A. THE THREE-MODULUS RSNS

The structure for a single-modulus residue sequence (MRS) in an *N*-modulus RSNS is

$$x_h = [0,0,\cdots,0,0,1,1,\cdots 1,1,\cdots,m,m,\cdots,m,m,\cdots,1,1,\cdots 1,1]. \tag{4.1}$$

Each integer residue is repeated *N* times, forming a sequence with length $2mN$, where *m* is the modulus [28]. For the three-modulus case, the RSNS modulus residue sequence and corresponding index *h* are shown in Figure 4.1. The length of the sequence is 6*m*.

x_h	0	0	0	1	1	1	…	m	m	m	$(m-1)$	$(m-1)$	$(m-1)$	…	1	1	1
h	0	1	2	3	4	5	…	$3m$	$3m+1$	$3m+2$	$3m+3$	$3m+4$	$3m+5$	…	$6m-3$	$6m-2$	$6m-1$

Figure 4.1 Modulus residue sequence for a three-modulus RSNS.

When forming the RSNS, all moduli are required to be pair-wise relatively prime (PRP). Once the PRP moduli are chosen, each MRS is fashioned according to Figure 4.1. Next, the MRS for one modulus is circularly shifted left one position while the MRS for a second MRS is circularly shifted left two positions relative to the non-shifted MRS. Finally, the three MRSs are repeated a number of times and stacked in modulus-index order to form vectors. Figure 4.2 shows a portion of the three-modulus RSNS structure for the moduli $m_i = [3\ \ 4\ \ 5]$.

X_h	$(m_1 = 3)$	0	0	0	1	1	1	2	2	2	3	3	3	2	2	2	1	1	1	0	0	0	1	1	1	…
	$(m_2 = 4)$	0	0	1	1	1	2	2	2	3	3	3	4	4	4	3	3	3	2	2	2	1	1	1	0	…
	$(m_3 = 5)$	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	4	4	4	3	3	3	2	2	…
	h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…

Figure 4.2 Three-modulus RSNS structure.

The Gray-code properties can be clearly seen by noticing that only a single MRS transitions to the next integer at each position. Figure 4.3 shows a plot of the residues from Figure 4.2.

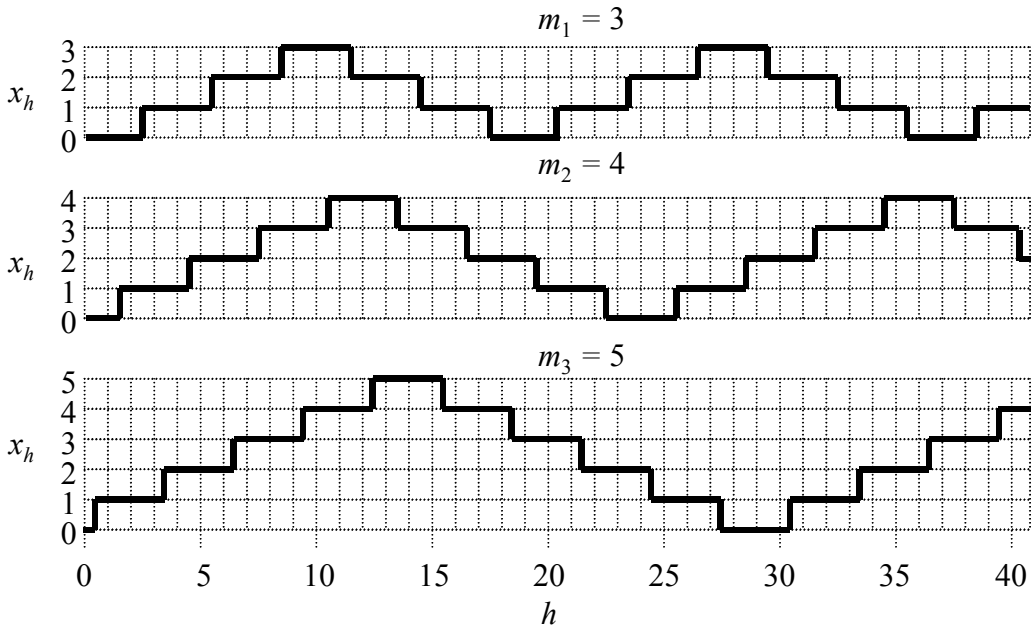

Figure 4.3 Plotting the residues of a three-modulus RSNS yields a folded stair-step structure.

Plotting the residues for each MRS versus the index h yields a folded, stair-step structure which clearly shows the result of the left-shift in the second and third MRSs. For consistency, the analysis in this research assumes that the MRS corresponding to moduli m_1, m_2, and m_3 are always circularly left-shifted by 0, 1, and 2 positions, respectively, and that $m_1 < m_2 < m_3$. It has been shown that the *size* of $\hat{M}$ in any particular RSNS is independent of which MRS receives the left-shifts. However, the *location* of $\hat{M}$ is affected by the choice of MRS that receives the left-shifts [30]. The fundamental period for the three-modulus RSNS is [30]

$$P_f = 6m_1m_2m_3 . \tag{4.2}$$

The i^{th} MRS has a length, or *folding period*, of $6m_i$ that is repeated exactly M/m_i times in the RSNS fundamental period where M is the product of the system moduli and m_i is the modulus of the i^{th} MRS. The remainder of this chapter is original work on the analysis of the three-modulus and N-modulus RSNS and formulation of fundamental analytic equations describing the structure of the RSNS.

There are three fundamental types of redundancies in each MRS as illustrated by Figure 4.4. First, since the MRS is periodic there exist redundancies from period to period, which are defined in this dissertation as *Type 0 redundancies*. In addition, the symmetry of the MRS (illustrated by the fold in the plot above) creates a redundancy on the rise of the fold and on the fall of the fold. These redundancies exist within a single MRS and are defined in this dissertation as *Type 1 redundancies*. Finally, the residues are repeated three times each (causing the stair step appearance of the plot) and are defined as *Type 2 redundancies*.

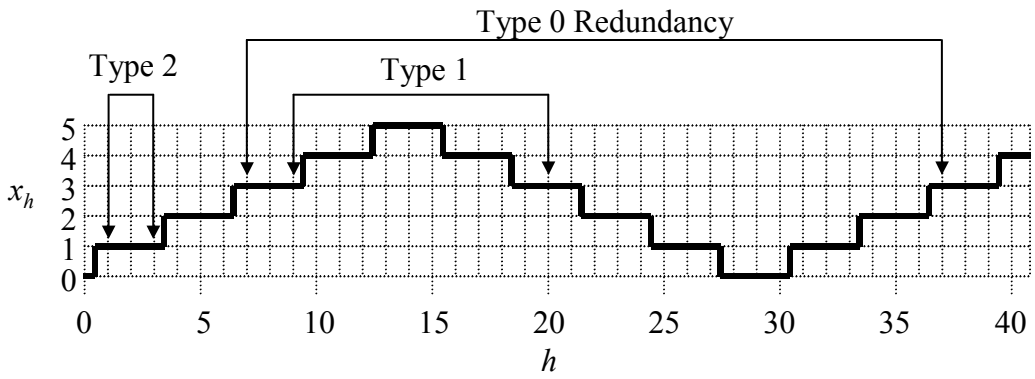

Figure 4.4 Single MRS redundancy types ($m = 5$).

Formulation of a straightforward analytical expression for the MRS redundancies is difficult because of the stair-step form of the MRS plot. The task is made simpler, however, by decimating each MRS into three sub-sequences. Each modulus residue sub-sequence (MRSS) is composed of values from the original MRS at positions where $h \equiv 0 \pmod 3$, $h \equiv 1 \pmod 3$, and $h \equiv 2 \pmod 3$. Figure 4.5 illustrates decimating an MRS ($m_1 = 3$) into three MRSSs.

$m_1 = 3$	x_h	0	0	0	1	1	1	2	2	2	3	3	3	2	2	2	1	1	1	0	0	0	1	…
$h = 0 \pmod 3$	x_h	0			1			2			3			2			1			0			1	…
$h = 1 \pmod 3$	x_h		0			1			2			3			2			1			0			…
$h = 2 \pmod 3$	x_h			0			1			2			3			2			1			0		…
	h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	…

Figure 4.5 A single MRS decimated into three MRSSs.

The top row in the figure is the original MRS and below it are the three MRSSs and finally the position index h. Notice that the positions of the MRSS integers do not align with each other. Every integer value h falls within one and only one MRSS. Figure 4.6 provides a plot of the three MRSSs in each MRS for the $m_i = [3\ \ 4\ \ 5]$ case.

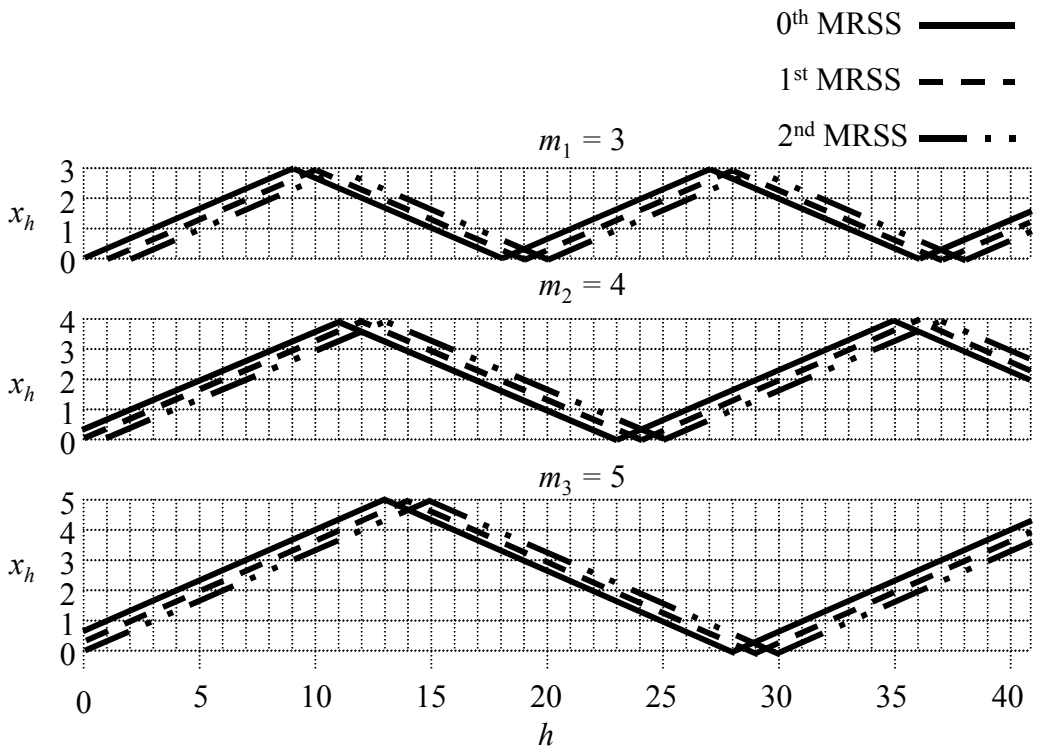

Figure 4.6 Three-modulus RSNS MRSS plot.

By forming the three MRSSs, the Type 2 redundancy is eliminated. However, all redundancies in each of the three MRSSs must be analytically described instead. Since the three MRSSs in each MRS are mutually exclusive in terms of the position index *h*, corresponding MRSSs can be analyzed as a group across the three main MRSs. Furthermore, the MRSSs shown in Figure 4.6 do not have the stair-step structure of the main MRS in Figure 4.3, but rather have a form similar to the well-understood folded SNS structure described in [36]. As a result, familiar methods of solving *N*-modulus SNS can be applied to the MRSS systems.

Since Type 2 redundancies do not exist in the MRSSs, all redundancies must be either Type 0 or Type 1. Following the methodology in [28] and looking at the parity of the residues forming the vector X_h (e = even, o = odd) provides insight into the minimum separation between redundant vectors. Figure 4.7 shows a RSNS even/odd periodicity of length six for the three-modulus RSNS example and therefore the distance between redundant vectors must always be a multiple of six.

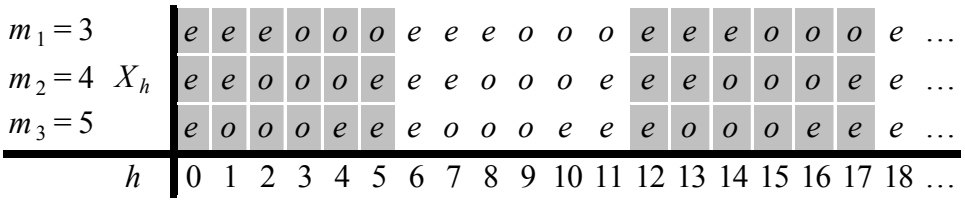

Figure 4.7 Parity of residue vectors.

Redundant vectors at positions *h* and $h+k$ are written $X_h = X_{h+k}$. Thus, from the analysis above, *k* must be a multiple of six for the three-modulus RSNS.

Next, the basic redundancies for individual MRSs are defined. Given that Type 0 redundancies are redundancies that occur at multiples of the MRS period, 6*m*, an expression for the position of the redundancy for a single MRS such that $x_h = x_{h+k}$ is given by

$$k \equiv 0 \pmod{6m}. \tag{4.3}$$

Note this redundancy is independent of the starting position *h*. Type 1 redundancies are redundancies that occur *within* an MRS period due to the symmetry of the MRS. An expression for the position of the Type 1 redundancy for a single MRS such that $x_h = x_{h+k}$ is given by

$$h + (h+k) \equiv 0 \pmod{6m}, \text{ if } h \equiv 0 \pmod{3} \quad (0^{\text{th}} \text{ MRSS}), \tag{4.4}$$

$$h + (h+k) \equiv 2 \pmod{6m}, \text{ if } h \equiv 1 \pmod{3} \quad (1^{\text{st}} \text{ MRSS}), \tag{4.5}$$

and

$$h + (h+k) \equiv 4 \pmod{6m}, \text{ if } h \equiv 2 \pmod{3} \quad (2^{\text{nd}} \text{ MRSS}). \tag{4.6}$$

Note that this redundancy is dependent on the start position *h* as well as the sub-sequence number (0^{th}, 1^{st}, or 2^{nd} MRSS). The MRSS number is also referred to as the *MRSS index*.

Equations (4.3)-(4.6) are extensions of the two-modulus RSNS results given in [28]. Furthermore, since Type 0 redundancies occur between different folding periods and Type 1 redundancies occur inside of a single folding period, they are mutually exclusive and any redundancy must be either Type 0 or Type 1, but not both. Therefore, to find all redundancies in the fundamental period for the three-modulus RSNS, it is neces-

sary to find all combinations of both Type 0 and Type 1 redundancies for all three MRSs. In other words, there is a redundant vector pair in the three-modulus RSNS wherever all three of the individual MRS redundancies (whether Type 0 of Type 1) align. Since there are three MRSs each with two possible types of redundancies (Type 0 and Type 1), there are 2^3 or 8 possible redundancy permutations. All of the possible redundancy permutations are summarized in the rows of the table in Figure 4.8 as separate cases. All cases that have at least one Type 1 redundancy have three sub-cases because Type 1 redundancies have three MRSSs.

Case Label	**MRS 3 Redundancy Type**	**MRS 2 Redundancy Type**	**MRS 1 Redundancy Type**	**Combination Decimal Value**	**Ordered Decimal Value Becomes Combination Number (second label digit)**
Case 010	Type 0	Type 0	Type 0	$000_2 = 0$	1
Case 110 Case 111 Case 112	Type 0	Type 0	Type 1	$001_2 = 1$	1
Case 120 Case 121 Case 122	Type 0	Type 1	Type 0	$010_2 = 2$	2
Case 130 Case 131 Case 132	Type 1	Type 0	Type 0	$100_2 = 4$	3
Case 210 Case 211 Case 212	Type 0	Type 1	Type 1	$011_2 = 3$	1
Case 220 Case 221 Case 222	Type 1	Type 0	Type 1	$101_2 = 5$	2
Case 230 Case 231 Case 232	Type 1	Type 1	Type 0	$110_2 = 6$	3
Case 310 Case 311 Case 312	Type 1	Type 1	Type 1	$111_2 = 7$	1

Figure 4.8 Table of redundancy types for three-modulus RSNS.

The labeling of the cases is different than in [28] due to the increase in the number of cases to be analyzed. Furthermore, for systems with more than three MRSs, the number of cases increases so dramatically that a logical case labeling system must be developed that is suitable for implementation in a computer algorithm. For consistency, each case is labeled with three identifier digits, as shown in the first column of Figure 4.9.

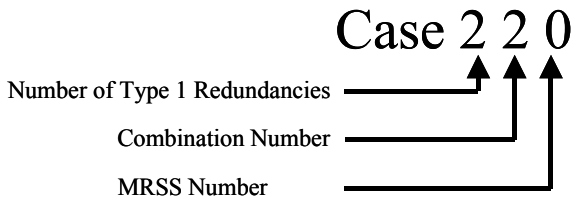

Figure 4.9 Case label example.

The first digit in the case label represents the number of MRSs that contain a Type 1 redundancy, and ranges from zero to three for the three-modulus RSNS. For example, in the three-modulus RSNS, all Case 1XX redundancies have only one MRS with a Type 1 redundancy and therefore the two remaining MRSs must be Type 0 redundancies. The second digit in the case label represents the particular assignment of Type 0 and Type 1 redundancies to specific MRSs. For instance, referring to the second row of Figure 4.8, the first combination of Case 1XX shows that MRS two and MRS three have a Type 0 redundancy, and MRS one has a Type 1 redundancy. Moreover, omitting the word "Type" before each redundancy, the combinations form binary numbers whose decimal value is given in the fourth column of Figure 4.8, sorted smallest to largest within each major case grouping indicated in the figure by the double separator lines. Sequentially numbering the combinations within each major case group as shown in the last column in the figure results in the unique *combination number* for each case group and is the second digit in each case label. Lastly, the third digit in the case label represents the MRSS index and ranges from zero to two for the three-modulus RSNS. An illustrative example follows. For the case label in Figure 4.9, Case 220, a two as the first digit specifies that there are two Type 1 redundancies (and therefore one Type 0 redundancy) for the three MRSs. The two as the second digit signifies that the particular order

of the redundancies is the second largest binary value ($101_2 = 5$). The zero in the third digit of the example case label indicates the redundancies are computed for the 0^{th} MRSS only.

Once the redundancy types have been defined and the number of cases has been established, it is necessary to form the congruence equations for each MRS in the three-modulus RSNS. Once the congruence equations are formed for each individual MRS, they will be combined in the next section into systems of congruence equations and solved for the locations of the redundant vectors.

For the first MRS, where $m = m_1$, rearranging the single MRS congruence equations (4.3)-(4.6) yields

$$\frac{k}{6} \equiv 0 \pmod{m_1} \tag{4.7}$$

for Type 0 redundancies and

$$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_1}, \quad \text{if} \quad h \equiv 0 \pmod 3 \quad (0^{\text{th}} \text{ MRSS}), \tag{4.8}$$

$$\frac{k}{6} \equiv -\frac{(h-1)}{3} \pmod{m_1}, \quad \text{if} \quad h \equiv 1 \pmod 3 \quad (1^{\text{st}} \text{ MRSS}), \tag{4.9}$$

and

$$\frac{k}{6} \equiv -\frac{(h-2)}{3} \pmod{m_1}, \quad \text{if} \quad h \equiv 2 \pmod 3 \quad (2^{\text{nd}} \text{ MRSS}) \tag{4.10}$$

for Type 1 redundancies. For the second MRS, where $m = m_2$, rearranging the single MRS congruence equations (4.3)-(4.6) while applying a one position left shift ($h = h + 1$) yields

$$\frac{k}{6} \equiv 0 \pmod{m_2} \tag{4.11}$$

for Type 0 redundancies and

$$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_2}, \quad \text{if} \quad h \equiv 0 \pmod 3 \quad (0^{\text{th}} \text{ MRSS}), \tag{4.12}$$

$$\frac{k}{6} \equiv -\frac{(h-1)}{3} \pmod{m_2}, \quad \text{if} \quad h \equiv 1 \pmod 3 \quad (1^{\text{st}} \text{ MRSS}), \tag{4.13}$$

and

$$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_2}, \quad \text{if} \quad h \equiv 2 \pmod 3 \quad (2^{\text{nd}} \text{ MRSS}) \tag{4.14}$$

for Type 1 redundancies. For the third MRS, where $m = m_3$, rearranging the single MRS congruence equations (4.3)-(4.6) while applying a two position left shift ($h = h + 2$) yields

$$\frac{k}{6} \equiv 0 \pmod{m_3} \tag{4.15}$$

for Type 0 redundancies and

$$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_3}, \quad \text{if} \quad h \equiv 0 \pmod 3 \quad (0^{\text{th}} \text{ MRSS}), \tag{4.16}$$

$$\frac{k}{6} \equiv -\frac{(h+2)}{3} \pmod{m_3}, \quad \text{if} \quad h \equiv 1 \pmod 3 \quad (1^{\text{st}} \text{ MRSS}), \tag{4.17}$$

and

$$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_3}, \quad \text{if} \quad h \equiv 2 \pmod 3 \quad (2^{\text{nd}} \text{ MRSS}) \tag{4.18}$$

for Type 1 redundancies.

Equations (4.7) through (4.18) represent all possible single-MRS redundancies. Notice that the only unique elements of (4.8)-(4.10), (4.12)-(4.14), and (4.16)-(4.18) are the MRS moduli and the numerators of the symmetrical residues, which can be easily represented in matrix form. The first column of the matrix is composed of the numera-

tors of the symmetrical residues from (4.8)-(4.10), while the second column comes from (4.12)-(4.14), and the third column comes from (4.16)-(4.18), resulting in

$$\begin{bmatrix} h+0 & h+0 & h+0 \\ h-1 & h-1 & h+2 \\ h-2 & h+1 & h+1 \end{bmatrix}, \tag{4.19}$$

or simply

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & -1 & 2 \\ -2 & 1 & 1 \end{bmatrix}. \tag{4.20}$$

The MRS number indexes the columns of the matrix and the MRSS number indexes the rows. The simplified matrix in (4.20) shows a particular structure. Starting with the leftmost column, each successive column is the previous column circularly shifted toward the top of the matrix with each element incremented by one. This matrix structure and pattern is particularly useful for automatically generating the congruence equations in computer algorithms as well as efficiently describing systems with greater than three MRSs.

B. CASE-BY-CASE SOLUTION FOR THE REDUNDANCY LOCATIONS

While the previous section defined all redundancies for each individual MRS, this section combines the single-MRS congruence equations on a case-by-case basis into systems of congruence equations in order to find all of the vector redundancy locations. All combinations of single-MRS congruence equations must be computed following the table in Figure 4.8. Moreover, since the MRSSs are mutually exclusive, all Type 1 redundancies must be from the same MRSS for each case. Solving the resulting systems of congruence equations produces analytical solutions to the corresponding RSNS vector redundancy locations.

1. Case 010

For Case 010, only Type 0 redundancies are chosen for MRSs 1, 2, and 3 as shown in the first row of Figure 4.8. The expressions for the individual MRS Type 0 redundancies are given by (4.7), (4.11), and (4.15), which when combined yield the system

$$\begin{aligned} \frac{k}{6} &\equiv 0 \pmod{m_1}, \\ \frac{k}{6} &\equiv 0 \pmod{m_2}, \\ \frac{k}{6} &\equiv 0 \pmod{m_3}. \end{aligned} \tag{4.21}$$

The system in (4.21) has the solution that k is a multiple of $6m_1m_2m_3$. In other words, the solution to this case is the redundant vector that occurs at multiples of the fundamental period. This is not surprising since it coincides with the definition of the fundamental period. Since all redundancies in this case are Type 0, there are no sub-cases. The redundancies found in this case are uninteresting due to the fact that the problem is focused on finding redundancies *within* the fundamental period. Case 010 is useful because it introduces a solution common to all systems of equations with structure similar to (4.21) and is referenced in subsequent cases.

2. Case 31X

Cases 31X are considered next because they are relatively simple to solve and, like Case 010, have a fundamental solution that subsequent cases utilize. For the three-modulus RSNS, Case 31X represents the condition where the congruence equations for each MRS are all chosen from the Type 1 Redundancy column, corresponding to the last row of Figure 4.8. Because only Type 1 redundancies are chosen, there is only one Case 3XX combination (albeit with three sub-cases) to analyze. For Case 310, the system of congruence equations formed from the individual MRS redundancy expressions (4.8), (4.12), and (4.16) is

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_2}, \\ \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_3}, \end{aligned} \tag{4.22}$$

where each equation is chosen from the 0^{th} MRSS. The solution to (4.22) is a particularly simple Chinese Remainder Theorem (CRT) problem [28]. Since $\frac{k}{6} = -\frac{h}{3}$ is one solution, all solutions are $\frac{k}{6} = -\frac{h}{3} + a(m_1 m_2 m_3)$, where *a* is any integer. Rearranging and solving for *h* yields the form

$$h_{Case310} = a(3m_1 m_2 m_3) - \frac{k}{2}. \tag{4.23}$$

Therefore, a redundancy occurs at $h = h + k$, which is

$$h_{Case310} + k = a(3m_1 m_2 m_3) + \frac{k}{2}. \tag{4.24}$$

Since the Case 310 vector redundancies occur at $a(3m_1 m_2 m_3) \pm \frac{k}{2}$, multiples of $3m_1 m_2 m_3$ are defined as *centers of redundancy* (COR) [28].

For Case 311 and Case 312 (1^{st} and 2^{nd} MRSSs of Case 31X) the systems of congruence equations are

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{(h-1)}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv -\frac{(h-1)}{3} \pmod{m_2}, \\ \frac{k}{6} &\equiv -\frac{(h+2)}{3} \pmod{m_3} \end{aligned} \tag{4.25}$$

for Case 311, and

$$\begin{aligned}\frac{k}{6} &\equiv -\frac{(h-2)}{3} \pmod{m_1},\\ \frac{k}{6} &\equiv -\frac{(h+1)}{3} \pmod{m_2},\\ \frac{k}{6} &\equiv -\frac{(h+1)}{3} \pmod{m_3}\end{aligned} \tag{4.26}$$

for Case 312. After [28], the form of the solutions to (4.25) and (4.26) is exactly the same as the form of the 0^{th} MRSS solution (4.23) except their COR are shifted by a constants, $h_{shift31X}$, which are the least positive solutions (LPS) of the system of congruence equations

$$\begin{aligned}\frac{(h_{shift311}-1)}{3} &\equiv 0 \pmod{m_1},\\ \frac{(h_{shift311}-1)}{3} &\equiv 0 \pmod{m_2},\\ \frac{(h_{shift311}+2)}{3} &\equiv 0 \pmod{m_3}\end{aligned} \tag{4.27}$$

for Case 311, and

$$\begin{aligned}\frac{(h_{shift312}-2)}{3} &\equiv 0 \pmod{m_1},\\ \frac{(h_{shift312}+1)}{3} &\equiv 0 \pmod{m_2},\\ \frac{(h_{shift312}+1)}{3} &\equiv 0 \pmod{m_3}\end{aligned} \tag{4.28}$$

for Case 312. To see why this is so, let $h_{shift311}$ be the LPS to (4.27). Now, given that $h_{Case310}$ and $h_{Case310}+k$ is a redundancy for Case 310, $h_{Case310}+h_{shift311}$ and $h_{Case310}+h_{shift311}+k$ is a redundancy for Case 311 for the following reason. Letting $h = h_{Case310}+h_{shift311}$ for the Case 311 system of congruence equations in (4.25) yields the system

$$
\begin{aligned}
-\frac{\left(h_{Case310}+\ h_{shift311}\ -1\right)}{3} &= -\frac{\left(h_{shift311}-1\ \right)}{3}-\frac{h_{Case310}}{3} \equiv -\frac{h_{Case310}}{3} \equiv \frac{k}{6}(\text{mod } m_1),\\
-\frac{\left(h_{Case310}+\ h_{shift311}-1\right)}{3} &= -\frac{\left(h_{shift311}-1\ \right)}{3}-\frac{h_{Case310}}{3} \equiv -\frac{h_{Case310}}{3} \equiv \frac{k}{6}(\text{mod } m_2), \qquad (4.29)\\
-\frac{\left(h_{Case310}+\ h_{shift311}+2\right)}{3} &= -\frac{\left(h_{shift311}+2\ \right)}{3}-\frac{h_{Case310}}{3} \equiv -\frac{h_{Case310}}{3} \equiv \frac{k}{6}(\text{mod } m_3).
\end{aligned}
$$

Therefore, the translation by $h_{shift311}$ of the Case 310 redundancy results in the Case 311 redundancy. A similar proof can be formed for the Case 312 redundancy. The system of congruence equations in (4.27) and (4.28) can be solved using the general form of the CRT. After solving for $h_{shift311}$ and $h_{shift312}$, the solution to the location of the Case 311 redundancies becomes

$$h_{Case311} = \left(a(3m_1m_2m_3) + h_{shift311}\right) - \frac{k}{2}, \qquad (4.30)$$

while the solution to the Case 312 redundancies becomes

$$h_{Case312} = \left(a(3m_1m_2m_3) + h_{shift312}\right) - \frac{k}{2}. \qquad (4.31)$$

This represents all solutions to all Case 31X redundancies for the three-modulus RSNS system.

3. Case 1XX

Case 1XX redundancies represent redundant RSNS vectors with individual MRS elements composed of one Type 1 and two Type 0 redundancies. Contrary to the previous two cases considered, there are *three* different ways to combine one Type 1 and two Type 0 redundancies. The three combinations are shown in rows two, three, and four of Figure 4.8. For Case 110, the system of congruence equations formed from (4.8), (4.11), and (4.15) is

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv 0 \pmod{m_2}, \\ \frac{k}{6} &\equiv 0 \pmod{m_3}. \end{aligned} \tag{4.32}$$

The bottom two equations in (4.32) have the same form as the three equations in (4.21) from Case 010 and the resulting solution is that k must be a multiple of $6m_2m_3$. Substituting this solution for k into the top equation in the system in (4.32) yields

$$-m_2m_3 \equiv \frac{h}{3} \pmod{m_1}, \tag{4.33}$$

which has the solution

$$h_{\text{Case 110}} = -3m_2m_3 + a(3m_1), \tag{4.34}$$

where a is any integer. This solution is interpreted as follows: there is a Case 110 vector redundancy at every position h and $h+k$ where k is a multiple of $6m_2m_3$ and $h = -3m_2m_3$ plus any integer multiple of $3m_1$. Comparing (4.34) and (4.23), it is apparent that $3m_1$ is a COR for Case 110 redundancies.

For Case 111, the system of congruence equations formed from (4.9), (4.11), and (4.15) is

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{(h-1)}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv 0 \pmod{m_2}, \\ \frac{k}{6} &\equiv 0 \pmod{m_3}. \end{aligned} \tag{4.35}$$

The congruence equations in (4.35) are the same as (4.32) except for the top equation, which is chosen from the 1st MRSS of MRS 1 rather than the 0th MRSS. Examining the similarities of (4.32) and (4.35), it is clear that if there is a redundancy at positions $h_{Case110}$ and $h_{Case110}+k$ there is also a redundancy at $h_{Case111}$ and $h_{Case111}+k$, with

$h_{\text{Case111}} = h_{\text{Case110}} + 1$. The proof of which is simply to substitute $h_{\text{Case111}} = h_{\text{Case110}} + 1$ into the Case 111 system of congruence equations given in (4.35) and see that the result is exactly the Case 110 system of congruence equations given in (4.32).

For Case 112, the system of congruence equations formed from (4.10), (4.11), and (4.15) is

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{(h-2)}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv 0 \pmod{m_2}, \\ \frac{k}{6} &\equiv 0 \pmod{m_3}. \end{aligned} \tag{4.36}$$

The congruence equations in (4.36) are the same as (4.32) except for the top equation, which is chosen from the 2nd MRSS of MRS 1 rather than the 0th MRSS. As in Case 111, the solution to Case 112 redundancies is a shifted version of the 0th MRSS solution, so there is a redundancy at positions $h_{Case112}$ and $h_{Case112} + k$, with $h_{\text{Case112}} = h_{\text{Case110}} + 2$. Thus, all redundancies have been identified and analytic solutions provided for all Case 11X redundancies.

According to Figure 4.8, there are two more Case 1XX combinations (Case 12X and Case 13X) corresponding to rows three and four. Fortunately, they are solved in the same way as Case 110, Case 111, and Case 112. For completeness, the systems of congruence equations for Case 12X including all MRSSs are shown in Figure 4.10.

Case 120	Case 121	Case 122
$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_2}$	$\frac{k}{6} \equiv -\frac{(h-1)}{3} \pmod{m_2}$	$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_2}$
$\frac{k}{6} \equiv 0 \pmod{m_3}$	$\frac{k}{6} \equiv 0 \pmod{m_3}$	$\frac{k}{6} \equiv 0 \pmod{m_3}$

Figure 4.10 Systems of congruence equations for Case 12X.

The solution, found in the same manner as Case 11X, is that for all MRSSs, k is a multiple of $6m_1m_3$ and

$$\begin{aligned} h_{Case120} &= -3m_1m_3 + a(3m_2), \\ h_{Case121} &= h_{Case120} + 1, \\ h_{Case122} &= h_{Case120} - 1. \end{aligned} \tag{4.37}$$

Similarly, the systems of congruence equations for Case 13X including all MRSSs are shown in Figure 4.11.

Case 130	Case 131	Case 132
$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$
$\frac{k}{6} \equiv 0 \pmod{m_2}$	$\frac{k}{6} \equiv 0 \pmod{m_2}$	$\frac{k}{6} \equiv 0 \pmod{m_2}$
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+2)}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_3}$

Figure 4.11 Systems of congruence equations for Case 13X.

The solution, found in the same manner as Case 11X, is that for all MRSSs, k is a multiple of $6m_1m_2$ and

$$\begin{aligned} h_{Case130} &= -3m_1m_2 + a(3m_3), \\ h_{Case131} &= h_{Case130} - 2, \\ h_{Case132} &= h_{Case130} - 1. \end{aligned} \tag{4.38}$$

Placing the unique elements of the solutions for all Case 1XX equations in a matrix with (4.34) in the first column, (4.37) in the second column, and (4.38) in the third column yields

$$\begin{bmatrix} h_{Case110} & h_{Case120} & h_{Case130} \\ h_{Case110} + 1 & h_{Case120} + 1 & h_{Case130} - 2 \\ h_{Case110} + 2 & h_{Case120} - 1 & h_{Case130} - 1 \end{bmatrix}, \tag{4.39}$$

or simply

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & -2 \\ 2 & -1 & -1 \end{bmatrix}, \tag{4.40}$$

where the columns represent the MRSs and the rows represent the MRSSs. The interesting pattern that emerges is that the matrix columns of (4.40) are simply the negatives of the circularly shifted and incremented columns found in (4.20). Again, this matrix structure is particularly useful for generating the solutions for systems of congruence equations in computer algorithms as well as calculating redundancy locations for systems with greater than three MRSs. Since the matrix given in (4.40) is the same for every three-modulus RSNS regardless of the choice of moduli, once the 0^{th} MRSS solution is found for each Case 1XX combination, the 1^{st} and 2^{nd} MRSS solutions can be found simply by shifting the 0^{th} MRSS solution by the values found in (4.40), thereby rendering the computation for the remaining MRSSs unnecessary. This completes the identification and analytic solution to all Case 1XX redundancies.

4. Case 2XX

Case 2XX redundancies represent redundant RSNS vectors whose individual MRS elements are composed of one Type 0 and two Type 1 redundancies. Similar to the previous case, there are three different ways to combine one Type 0 and two Type 1 redundancies. The three combinations are shown in rows five, six, and seven of Figure 4.8. For Case 210, the system of congruence equations formed from (4.8), (4.12), and (4.15) is

$$\begin{aligned} \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_1}, \\ \frac{k}{6} &\equiv -\frac{h}{3} \pmod{m_2}, \\ \frac{k}{6} &\equiv 0 \pmod{m_3}. \end{aligned} \tag{4.41}$$

The consequence of the third equation in (4.41) is simply that k is a multiple of $6m_3$. The remaining two equations are solved in a manner similar to the Case 310 equations and therefore have the solution

$$h_{Case210} = a(3m_1m_2) - \frac{k}{2}, \tag{4.42}$$

with the $h = h + k$ redundancy at

$$h_{Case210} + k = a(3m_1m_2) + \frac{k}{2}. \tag{4.43}$$

Analogous to Case 310, a Case 210 COR is discovered at $3m_1m_2$. In addition, since it is known that k is a multiple of $6m_3$, by substituting the least positive k into (4.43) the solution above can also be expressed as

$$h_{Case210} = a(3m_1m_2) - 3m_3, \tag{4.44}$$

with the $h = h + k$ redundancy at

$$h_{Case210} + k = a(3m_1m_2) + 3m_3. \tag{4.45}$$

The solutions for the other two sub-cases (Case 211 and Case 212) are found exactly like Case 311 by simply shifting the Case 210 COR by a constant equal to the LPS to the following systems of congruence equations. For Case 211, $h_{shift211}$ is the LPS to

$$\begin{aligned} \frac{\left(h_{shift211} - 1\right)}{3} &\equiv 0 \pmod{m_1}, \\ \frac{\left(h_{shift211} - 1\right)}{3} &\equiv 0 \pmod{m_2}, \end{aligned} \tag{4.46}$$

which can be solved using the general form of the CRT. The resulting solution to the Case 211 vector redundancy is

$$h_{Case211} = \left(a(3m_1m_2) + h_{shift211}\right) - \frac{k}{2}, \tag{4.47}$$

with the $h = h + k$ redundancy at

$$h_{Case211} + k = \left(a(3m_1m_2) + h_{shift211}\right) + \frac{k}{2}. \tag{4.48}$$

For Case 212, $h_{shift212}$ is the LPS to

$$\begin{aligned} \frac{\left(h_{shift212} - 2\right)}{3} &\equiv 0 \pmod{m_1}, \\ \frac{\left(h_{shift212} + 1\right)}{3} &\equiv 0 \pmod{m_2}, \end{aligned} \tag{4.49}$$

which can be solved using the general form of the CRT. The resulting solution to the Case 212 vector redundancy is

$$h_{Case212} = \left(a(3m_1m_2) + h_{shift212}\right) - \frac{k}{2}, \tag{4.50}$$

with the $h = h + k$ redundancy at

$$h_{Case212} + k = \left(a(3m_1m_2) + h_{shift212}\right) + \frac{k}{2}. \tag{4.51}$$

According to Figure 4.8, there are two more Case 2XX combinations (Case 22X and Case 23X) corresponding to rows six and seven of Figure 4.8. Fortunately, they are solved in the same way as Case 210, Case 211, and Case 212. For completeness, the systems of congruence equations for Case 22X including all MRSSs are shown in Figure 4.12.

Case 220	Case 221	Case 222
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_1}$	$\frac{k}{6} \equiv -\frac{(h-1)}{3} \pmod{m_1}$	$\frac{k}{6} \equiv -\frac{(h-2)}{3} \pmod{m_1}$
$\frac{k}{6} \equiv 0 \pmod{m_2}$	$\frac{k}{6} \equiv 0 \pmod{m_2}$	$\frac{k}{6} \equiv 0 \pmod{m_2}$
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+2)}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_3}$

Figure 4.12 Systems of congruence equations for Case 22X.

The solution, found in the same manner as Case 21X, is that for all MRSSs, k is a multiple of $6m_2$ and

$$\begin{aligned} h_{Case220} &= a(3m_1m_3) - \frac{k}{2}, \\ h_{Case221} &= \left(a(3m_1m_3) + h_{shift221}\right) - \frac{k}{2}, \\ h_{Case222} &= \left(a(3m_1m_3) + h_{shift222}\right) - \frac{k}{2}. \end{aligned} \tag{4.52}$$

The redundant vectors are located at $h + k$ for each MRSS and are

$$\begin{aligned} h_{Case220} + k &= a(3m_1m_3) + \frac{k}{2}, \\ h_{Case221} + k &= \left(a(3m_1m_3) + h_{shift221}\right) + \frac{k}{2}, \\ h_{Case222} + k &= \left(a(3m_1m_3) + h_{shift222}\right) + \frac{k}{2}. \end{aligned} \tag{4.53}$$

The shifts for the 1st and 2nd MRSSs are the LPS to the systems of congruence equations shown in Figure 4.13, which can be solved using the general form of the CRT. The systems of congruence equations for Case 23X including all MRSSs are shown in Figure 4.14.

Case 221 COR shift	Case 222 COR shift
$\frac{\left(h_{shift221} - 1\right)}{3} \equiv 0 \pmod{m_1}$	$\frac{\left(h_{shift222} - 2\right)}{3} \equiv 0 \pmod{m_1}$
$\frac{\left(h_{shift221} + 2\right)}{3} \equiv 0 \pmod{m_3}$	$\frac{\left(h_{shift222} + 1\right)}{3} \equiv 0 \pmod{m_3}$

Figure 4.13 Systems of congruence equations for Case 22X COR shifts.

Case 230	Case 231	Case 232
$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$	$\frac{k}{6} \equiv 0 \pmod{m_1}$
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_2}$	$\frac{k}{6} \equiv -\frac{(h-1)}{3} \pmod{m_2}$	$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_2}$
$\frac{k}{6} \equiv -\frac{h}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+2)}{3} \pmod{m_3}$	$\frac{k}{6} \equiv -\frac{(h+1)}{3} \pmod{m_3}$

Figure 4.14 Systems of congruence equations for Case 22X.

The solution, found in the same manner as Case 21X, is that for all MRSSs, k is a multiple of $6m_1$ and

$$\begin{aligned} h_{Case230} &= a(3m_2m_3) - \frac{k}{2}, \\ h_{Case231} &= \left(a(3m_2m_3) + h_{shift231}\right) - \frac{k}{2}, \\ h_{Case232} &= \left(a(3m_2m_3) + h_{shift232}\right) - \frac{k}{2}. \end{aligned} \tag{4.54}$$

The redundant vectors are located at $h+k$ for each MRSS are

$$\begin{aligned} h_{Case230} + k &= a(3m_2m_3) + \frac{k}{2}, \\ h_{Case231} + k &= \left(a(3m_2m_3) + h_{shift231}\right) + \frac{k}{2}, \\ h_{Case232} + k &= \left(a(3m_2m_3) + h_{shift232}\right) + \frac{k}{2}. \end{aligned} \tag{4.55}$$

The shifts for the 1st and 2nd MRSSs are the LPS to the systems of congruence equations shown in Figure 4.15, which can be solved using the general form of the CRT. This represents all solutions of the Case 2XX redundancies for the three-modulus RSNS.

Case 231 COR shift	Case 232 COR shift
$\frac{\left(h_{shift\,231}-1\right)}{3} \equiv 0 \pmod{m_2}$	$\frac{\left(h_{shift\,232}+1\right)}{3} \equiv 0 \pmod{m_2}$
$\frac{\left(h_{shift\,231}+2\right)}{3} \equiv 0 \pmod{m_3}$	$\frac{\left(h_{shift\,232}+1\right)}{3} \equiv 0 \pmod{m_3}$

Figure 4.15 Systems of congruence equations for Case 22X COR shifts.

5. Summary of Vector Redundancy Locations

The table in Figure 4.16 summarizes the solutions to the RSNS vector redundancy locations. It is interesting to note that all of the redundancies smaller than the fundamental period are symmetric around a *center of redundancy*. In addition, reading down the table the spacing between the redundant vectors (k) increases while the spacing between the COR decreases. Furthermore, the moduli used to calculate the value of k for each case are never the same moduli used to calculate the COR, and all of the moduli are used to calculate either one or the other. Of course, none of this is coincidence and all of these facts can be used to extend the three-modulus RSNS redundancy results to solve the more general N-modulus RSNS redundancies.

Case Label	Redundancies occur at h and $h+k$ where h is	and k is a multiple of	COR
Case 010	Any position in the fundamental period	$6m_1m_2m_3$	N/A
Case 110	$h_{\text{Case 110}} = a(3m_1) - \frac{k}{2}$	$6m_2m_3$	$a(3m_1)$
Case 111	$h_{\text{Case 111}} = \left(a(3m_1)+1\right) - \frac{k}{2}$		$a(3m_1)+1$
Case 112	$h_{Case\ 112} = \left(a(3m_1)+2\right) - \frac{k}{2}$		$a(3m_1)+2$
Case 120	$h_{Case120} = a(3m_2) - \frac{k}{2}$	$6m_1m_3$	$a(3m_2)$
Case 121	$h_{Case121} = \left(a(3m_2)+1\right) - \frac{k}{2}$		$a(3m_2)+1$
Case 122	$h_{Case122} = \left(a(3m_2)-1\right) - \frac{k}{2}$		$a(3m_2)-1$
Case 130	$h_{Case130} = a(3m_3) - \frac{k}{2}$	$6m_1m_2$	$a(3m_3)$
Case 131	$h_{Case131} = \left(a(3m_3)-2\right) - \frac{k}{2}$		$a(3m_3)-2$
Case 132	$h_{Case132} = \left(a(3m_3)-1\right) - \frac{k}{2}$		$a(3m_3)-1$
Case 210	$h_{Case210} = a(3m_1m_2) - \frac{k}{2}$	$6m_3$	$a(3m_1m_2)$
Case 211	$h_{Case211} = \left(a(3m_1m_2) + h_{shift211}\right) - \frac{k}{2}$		$a(3m_1m_2) + h_{shift211}$
Case 212	$h_{Case212} = \left(a(3m_1m_2) + h_{shift212}\right) - \frac{k}{2}$		$a(3m_1m_2) + h_{shift212}$
Case 220	$h_{Case220} = a(3m_1m_3) - \frac{k}{2}$	$6m_2$	$a(3m_1m_3)$
Case 221	$h_{Case221} = \left(a(3m_1m_3) + h_{shift221}\right) - \frac{k}{2}$		$a(3m_1m_3) + h_{shift221}$
Case 222	$h_{Case222} = \left(a(3m_1m_3) + h_{shift222}\right) - \frac{k}{2}$		$a(3m_1m_3) + h_{shift222}$
Case 230	$h_{Case230} = a(3m_2m_3) - \frac{k}{2}$	$6m_1$	$a(3m_2m_3)$
Case 231	$h_{Case231} = \left(a(3m_2m_3) + h_{shift231}\right) - \frac{k}{2}$		$a(3m_2m_3) + h_{shift231}$
Case 232	$h_{Case232} = \left(a(3m_2m_3) + h_{shift232}\right) - \frac{k}{2}$		$a(3m_2m_3) + h_{shift232}$
Case 310	$h_{Case310} = a(3m_1m_2m_3) - \frac{k}{2}$	6	$a(3m_1m_2m_3)$
Case 311	$h_{Case311} = \left(a(3m_1m_2m_3) + h_{shift311}\right) - \frac{k}{2}$		$a(3m_1m_2m_3) + h_{shift311}$
Case 312	$h_{Case312} = \left(a(3m_1m_2m_3) + h_{shift312}\right) - \frac{k}{2}$		$a(3m_1m_2m_3) + h_{shift312}$

Figure 4.16 Three-modulus RSNS redundancy summary table.

C. THE *N*-MODULUS RSNS

1. *N*-modulus RSNS Redundancy Analysis

The three-modulus RSNS redundancy model has repeatable and exploitable structure that can be extrapolated to form the general *N*-modulus RSNS congruence equations and solutions. Starting with the single MRS redundancies, the three-modulus RSNS congruence equations for a single MRS given in (4.3) through (4.6) can easily be extended to the *N*-modulus RSNS. Given that Type 0 redundancies are redundancies that occur at multiples of the MRS period, which is $2Nm$ for the *N*-modulus case, an expression for the position of the redundancy for a single MRS such that $x_h = x_{h+k}$ is given by

$$k \equiv 0 \pmod{2Nm}. \tag{4.56}$$

Note that this redundancy is not dependent on the starting position h. Type 1 redundancies are redundancies that occur *within* an MRS period due to the symmetry of the MRS. Expressions for the position of the *N*-modulus Type 1 redundancies for all MRSSs in single MRS such that $x_h = x_{h+k}$ is given by

$$\begin{aligned}
&h+(h+k) \equiv 0 \pmod{2Nm}, \text{ if } h \equiv 0 \pmod{N} \quad (0^{\text{th}} \text{ MRSS}),\\
&h+(h+k) \equiv 2 \pmod{2Nm}, \text{ if } h \equiv 1 \pmod{N} \quad (1^{\text{st}} \text{ MRSS}),\\
&h+(h+k) \equiv 4 \pmod{2Nm}, \text{ if } h \equiv 2 \pmod{N} \quad (2^{\text{nd}} \text{ MRSS}),\\
&\qquad\qquad\vdots\\
&h+(h+k) \equiv 2(N-1) \pmod{2Nm}, \text{ if } h \equiv N-1 \pmod{N} \quad (N^{\text{th}} \text{ MRSS}).
\end{aligned} \tag{4.57}$$

Note that this redundancy is dependent on the start position h as well as the MRSS number. To find all redundancies in the fundamental period for the *N*-modulus RSNS, it is necessary to find all combinations of both Type 0 and Type 1 redundancies for all N MRSs and associated MRSSs. In other words, there is a redundancy vector in the *N*-modulus RSNS wherever all N of the individual MRS redundancies (whether Type 0 of Type 1) align.

Computing the number and type of cases for *N*-modulus RSNS is only slightly more complicated than the three-modulus RSNS. The table in Figure 4.17 shows the combinations of Type 0 and Type 1 redundancies for three-, four-, and *N*-modulus RSNS. The three-modulus RSNS column of the table below corresponds to the rows in Figure

4.8, where there are 22 total cases and sub-cases organized into eight rows (combinations).

	Three-Modulus RSNS	**Four-Modulus RSNS**	***N*-Modulus RSNS**
Type 0/Type 1 Combinations	0 0 0 0 0 1 ⋮ 1 1 0 1 1 1	0 0 0 0 0 0 0 1 ⋮ 1 1 1 0 1 1 1 1	0 0 ⋯ 0 0 0 0 ⋯ 0 1 ⋮ 1 1 ⋯ 1 0 1 1 ⋯ 1 1
Number of combinations	8	16	2^N
Total number of combinations (including sub-cases)	22	61	$N\left(2^N-1\right)+1$

Figure 4.17 Table of RSNS redundancy combinations.

There are 2^N distinct redundancy type combinations for the N-modulus case and all but the first combination (which contains all Type 0 redundancies) will have N sub-cases. This is due to the existence of the N MRSSs shown in (4.57). Thus, the expression for the total number of combinations in the N-modulus RSNS is given by

$$N\left(2^N-1\right)+1 = N2^N - N + 1 . \tag{4.58}$$

The Type 0/Type 1 combinations are grouped into cases and labeled according to the same rules as the three-modulus RSNS. As a quick review of the case label notation, the first digit in the case label represents the number of MRSs that contain a Type 1 redundancy, which ranges from zero to N for the N-modulus RSNS. For example, all Case 1XX redundancies have only one MRS with a Type 1 redundancy and the remaining MRSs are Type 0 redundancies. The second digit in the case label is the particular combination of Type 0 and Type 1 redundancies and represents the ranking (or order) of the corresponding decimal value. That is, the combinations for each case are numbered se-

quentially starting with the combination with the smallest decimal value. The third digit in the case label represents the sub-case corresponding to the particular MRSS index within the set of N-modulus Type 1 redundancy congruence equations (4.57). The third digit is in the range of zero to $N-1$. Since the MRSSs are mutually exclusive, all Type 1 redundancies must be from the same MRSS for each sub-case.

Computing the number of combinations in each case requires the use of the binomial coefficient formula. Figure 4.18 shows the computation of the number of combinations for each N-modulus case.

$$\text{Number of Case 0XX: } \binom{N}{0} = 1$$

$$\text{Number of Case 1XX: } \binom{N}{1} = N$$

$$\text{Number of Case 1XX: } \binom{N}{2} = \frac{N!}{2(N-2)!}$$

$$\vdots$$

$$\text{Number of Case } N\text{XX: } \binom{N}{N} = 1$$

$$\text{where} \quad \binom{N}{k} = \frac{N!}{k!(N-k)!}$$

Figure 4.18 Grouping N-modulus RSNS combinations into cases.

Like the three-modulus RSNS, the particular form of each individual MRS and associated MRSSs must be individually defined. The form of the i^{th} MRS Type 0 redundancy is given by

$$\frac{k}{2N} \equiv 0 \pmod{m_i}, \tag{4.59}$$

which is simply a generalization of (4.7), (4.11), and (4.15). For the Type 1 i^{th} MRS congruence equations, the three-modulus matrix structures in (4.19) and (4.20) have a structure that can be extended to form the N-modulus Type 1 equations. The resulting N-modulus matrix structure used to form the Type 1 i^{th} MRS congruence equations is

$$\begin{bmatrix} h+0 & h+0 & h+0 & \cdots & h+0 \\ & \square & \square & \vdots & \iddots \\ h-1 & h-1 & h-N+1 & \cdots & h+(N-1) \\ & \square & \vdots \ \square & \vdots & \iddots \quad \vdots \\ h-2 & h-N & h+2 & \cdots & h+2 \\ \vdots & \square & \vdots \ \square & & \square \\ h-(N-1) & h+1 & h+1 & \cdots & h+1 \end{bmatrix}, \tag{4.60}$$

or simply

$$\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ & & \vdots & & \\ -1 & -1 & -N+1 & \cdots & N-1 \\ & \vdots & \vdots & & \vdots \\ -2 & -N & 2 & \cdots & 2 \\ \vdots & \vdots & & & \\ -(N-1) & 1 & 1 & \cdots & 1 \end{bmatrix}, \tag{4.61}$$

where the column index is the MRS number and the row index is the MRSS index. Like the three-modulus RSNS and starting with the leftmost column, each successive column is the previous column circularly shifted toward the top of the matrix with each element incremented by one as shown by the arrows in (4.60). Recall from the three-modulus case that the entries in the matrix are the numerators for the residues in the system of congruence equations. Consequently, using the pattern in the columns of the matrix, the expression for the N-modulus Type 1 redundancies for a single MRS is given by

$$\begin{aligned} &\frac{k}{2N} \equiv -\frac{h}{N} \pmod{m_i}, && \textit{if} \quad h \equiv 0 \pmod{N}, \\ &\frac{k}{2N} \equiv -\frac{(h-1)}{N} \pmod{m_i}, && \textit{if} \quad h \equiv 1 \pmod{N}, \\ &\quad \vdots \\ &\frac{k}{2N} \equiv -\frac{(h+i-1)}{N} \pmod{m_i}, && \textit{if} \quad h \equiv N-i+1 \pmod{N}, \\ &\quad \vdots \\ &\frac{k}{2N} \equiv -\frac{(h+1)}{N} \pmod{m_i}, && \textit{if} \quad h \equiv N-1 \pmod{N}. \end{aligned} \tag{4.62}$$

Equations (4.59) and (4.62) represent all redundancies for each *individual* MRS including all MRSSs for the *N*-modulus RSNS.

In order to find the *vector* redundancy locations for the *N*-modulus RSNS, the single-MRS congruence equations for all cases must be combined into systems of congruence equations. Solving the resulting systems of congruence equations will produce expressions for the *N*-modulus RSNS vector redundancy locations.

2. Case-by-Case Solution for the *N*-Modulus Redundancy Locations

The previous section defined the redundancies present in each *individual* MRS of an *N*-modulus RSNS. In order to find the *vector* redundancies in the *N*-modulus number system, the individual MRSs must be combined. The vector redundancies occur wherever individual MRS redundancies align across all N MRSs. The study of the *N*-modulus redundancy cases in this section follows the same process as the previously analyzed three-modulus system. First, an even/odd analysis of the *N*-modulus RSNS vectors similar to that performed in Figure 4.7 reveals that the *N*-modulus vector repetitions occur at a minimum distance of $2N$. Next, the following sections provide analytic expressions for the vector redundancy locations for each case in Figure 4.18.

a. Case 010

Like the three-modulus RSNS, there is just one *N*-modulus Case 0XX, labeled Case 010, and it has no MRSSs. The form of Case 010 is given by

$$\begin{aligned} \frac{k}{2N} &\equiv 0 \pmod{m_1}, \\ \frac{k}{2N} &\equiv 0 \pmod{m_2}, \\ &\vdots \\ \frac{k}{2N} &\equiv 0 \pmod{m_N}, \end{aligned} \tag{4.63}$$

which always has the solution that k is a multiple of

$$P_f = 2N\prod_i m_i \,, \tag{4.64}$$

so long as m_i are PRP. In other words, like the three-modulus RSNS there is a redundancy at multiples of the fundamental period. Again, the redundancies found in this case are uninteresting due to the fact that the problem is finding redundancies *within* a fundamental period.

b. Case N1X

Cases N1X are considered next because they are relatively simple to solve. The three-modulus RSNS Case 31X is equivalent to Case N1X for the N-modulus RSNS. Fortunately, it is solved in a similar manner. For the N-modulus RSNS, Case N1X represents the condition where the congruence equations for each MRS are all chosen from the Type 1 redundancy equations (4.62). Because only Type 1 redundancies are chosen, there is only one Case NXX combination to analyze (with N sub-cases). For Case N10, all congruence equations are formed from the top line of (4.62), which is the 0^{th} MRSS and have the form

$$\frac{k}{2N} \equiv -\frac{h}{N} \,(\text{mod}\, m_i)\,. \tag{4.65}$$

All solutions are given by

$$h_{CaseN10} = a\left(N\prod_{n=1}^{N} m_n\right) - \frac{k}{2}, \tag{4.66}$$

with the $h = h + k$ redundancy at

$$h_{CaseN10} + k = a\left(N\prod_{n=1}^{N} m_n\right) + \frac{k}{2}, \tag{4.67}$$

where k is a multiple of $2N$. For Case N10 there are exactly two COR: one at zero and one at $P_f/2$. Like Case 31X in the three-modulus RSNS, the remaining solutions for the $N-1$ MRSSs of Case N1X are shifted forms of the Case N10 solution and are given by

$$h_{CaseN1X} = \left(a\left(N\prod_{n=1}^{N} m_n \right) + h_{shiftN1X} \right) - \frac{k}{2}, \tag{4.68}$$

with the $h = h + k$ redundancy at

$$h_{CaseN1X} + k = \left(a\left(N\prod_{n=1}^{N} m_n \right) + h_{shiftN1X} \right) + \frac{k}{2}. \tag{4.69}$$

Furthermore, the shift ($h_{shiftN1X}$) of the COR in the equations above for each MRSS is computed by finding the LPS to the congruence equations formed in the same manner as the three-modulus RSNS (4.27) and (4.28). The resulting systems of equations can be solved using the general form of the CRT. Repeating this method for each sub-case, all Case *N*1X redundancies can be found for the *N*-modulus RSNS.

c. Case 1XX

Case 1XX redundancies represent redundant RSNS vectors with elements comprised of redundant residues that are all Type 0 except for one that is of Type 1. There are only *N* such combinations, as shown in the second line in Figure 4.18 (read as *N* choose 1). The Case 1XX three-modulus RSNS expressions can be extended to produce the *N*-modulus Case 1XX solutions. For Case 1X0, all but one of the congruence equations for the *N*-modulus RSNS come from (4.59) and will have the form

$$\frac{k}{2N} \equiv 0 \pmod{m_j}, \tag{4.70}$$

where the subscript *j* corresponds to the indices of all MRSs that have Type 0 redundancies. The final congruence equation is chosen from the 0^{th} MRSS of (4.62) and has the form

$$\frac{k}{2N} \equiv -\frac{h}{N} \pmod{m_i}, \tag{4.71}$$

where the subscript *i* corresponds to the channel that has the Type 1 redundancy. For this case only where there is only one Type 1 redundancy, the subscript *i* also represents the combination number (second case label digit). The simple solution to the set of congruence equations formed by (4.70) mandate that *k* must be a multiple of

$$2N\frac{M}{m_i}, \tag{4.72}$$

where

$$M = \prod_{n=1}^{N} m_n \, . \tag{4.73}$$

Substituting the least positive value for k into (4.71) results in the solution

$$h_{Case1X0} \;=\; a(m_i) - N\frac{M}{m_i}, \tag{4.74}$$

for Case 1X0 where the subscript i represents the combination number and ranges from 1 to N. For the solutions to the remainder of the $N-1$ MRSSs, Case 1X1 through Case 1X(N-1), the results of the three-modulus RSNS Case 1XX are again essential. The matrices in (4.39) show that once the 0th MRSS solution is found, the rest of the MRSSs are just shifted versions of the 0th MRSS solution. Therefore, the expression for the solution to the rest of the N-modulus Case 1XX redundancies is given by

$$h_{Case1is} \;=\; \left(a(m_i) + h_{shift1is}\right) - N\frac{M}{m_i}, \tag{4.75}$$

where i is the MRS index with the Type 1 redundancy and s is the MRSS index. The shifts for the COR ($h_{shift1is}$) are given by the matrix

$$h_{shift1is} \;\Leftrightarrow\; \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ & & \vdots & & \\ -1 & -1 & -N+1 & \cdots & N-1 \\ & \vdots & \vdots & & \vdots \\ -2 & -N & 2 & \cdots & 2 \\ \vdots & \vdots & & & \\ -(N-1) & 1 & 1 & \cdots & 1 \end{bmatrix}, \tag{4.76}$$

where i corresponds to the column and s corresponds to the row. This method produces all the solutions for the N-modulus Case 1XX redundancies.

d. Case 2XX through Case (N-1)XX

For the *N*-modulus RSNS systems, a generalization of the three-modulus Case 2XX analysis method can be used to provide solutions to all remaining *N*-modulus RSNS Case 2XX through Case (*N*-1)XX. For each case, the number of modulus combinations can be found by the binomial coefficient formula shown in Figure 4.18. Each modulus combination has *N* sub-cases. The congruence equations that represent the Type 0 redundancies determine the *length of the redundancy*. In other words, for each case, *k* is a multiple of

$$2N\prod_{j} m_j \ , \tag{4.77}$$

where *j* are the indices of all MRSs with Type 0 redundancies (e.g., there are $N-2$ MRSs with Type 0 redundancies for Case 2XX, $N-3$ MRSs with Type 0 redundancies for Case 3XX, etc.). The congruence equations that represent the Type 1 redundancies affect the *location of the COR*. The solution always has the form

$$h = \left(a(N\prod_{i} m_i) + h_{shiftXXX} \right) - \frac{k}{2}, \tag{4.78}$$

with the $h = h + k$ redundancy at

$$h + k = \left(a(N\prod_{i} m_i) + h_{shiftXXX} \right) + \frac{k}{2}, \tag{4.79}$$

where *i* are the indices of all MRSs with Type 1 redundancies and the unshifted COR is

$$N\prod_{i} m_i \ . \tag{4.80}$$

The COR shift $h_{shiftXXX}$ is the shift of the COR for Case XXX and is found by computing the LPS to the system of MRSS congruence equations formed only from the MRS Type 1 redundancy equations. Some examples of forming the system of MRSS congruence equations are given in (4.27), (4.28), (4.46), and Figure 4.13. The LPS to the MRSS system of congruence equations can be found using the general form of the CRT. The shift of the COR for all 0^{th} MRSSs are always zero. In other words, $h_{shiftXX0} = 0$. Using this

procedure, analytical expressions for all *N*-modulus RSNS Case 2XX through Case ($N-1$)XX redundancies can be produced.

3. Summary of *N*-Modulus Vector Redundancy Locations

The table in Figure 4.19 summarizes the *N*-modulus RSNS vector redundancy location solutions.

Case Label	Redundancies occur at h and $h+k$ where h is	and k is a multiple of	COR
Case 010	Any position in the fundamental period	$P_f = 2N\prod_{n=1}^{N} m_n$	None
Case 1X0	$h_{Case1X0} = a(m_i) - \frac{k}{2}$	$2N\frac{M}{m_i}$	$a(m_i)$
Case 1XX	$h_{Case1is} = \left(a(m_i) + h_{shift1is}\right) - \frac{k}{2}$		$a(m_i) + h_{shift1is}$
Case 2X0 ... Case (N-1)X0	$h = a\left(N\prod_i m_i\right) - \frac{k}{2}$	$2N\prod_j m_j$	$a\left(N\prod_i m_i\right)$
Case 2XX ... Case (N-1)XX	$h = \left(a\left(N\prod_i m_i\right) + h_{shiftXXX}\right) - \frac{k}{2}$		$a\left(N\prod_i m_i\right) + h_{shiftXXX}$
Case N10	$h_{CaseN10} = a\left(N\prod_{n=1}^{N} m_n\right) - \frac{k}{2}$	$2N$	$a\left(N\prod_{n=1}^{N} m_n\right)$
Case N1X	$h_{CaseN1X} = \left(a\left(N\prod_{n=1}^{N} m_n\right) + h_{shiftN1X}\right) - \frac{k}{2}$		$a\left(N\prod_{n=1}^{N} m_n\right) + h_{shiftN1X}$

Figure 4.19 *N*-modulus RSNS redundancy summary table.

The subscript *j* represents the MRSs with Type 0 redundancies, the subscript *i* represents the MRSs with Type 1 redundancies, and the parameter *s* is the MRSS index. All of the redundancies smaller than the fundamental period are symmetric around a *center of redundancy*. Notice that as the case number increases, the spacing between the redundant vectors (*k*) increases while the spacing between the COR decreases. Furthermore, the moduli used to calculate the value of *k* for each case are never the same moduli used to calculate the COR, and all of the moduli are used to calculate one or the other.

For the two-modulus RSNS, the vector redundancy expressions for four cases were combined to produce a closed-form analytical expression for $\hat{M}$ of certain moduli

classes [28]. That approach is not practical for any *N*-modulus RSNS other than the two-modulus case due to the exponential increase in the number of vector redundancy cases. However, the expressions for the three-modulus and *N*-modulus RSNS redundancies can be used to produce useful results as shown in the following sections.

D. $\hat{M}$ FOR MODULI $m-1,\ m,\ m+1$

Based on the analysis of the three-modulus RSNS, an analytic expression for $\hat{M}$ for the three-modulus RSNS with PRP moduli of the form $m_i = [m-1 \quad m \quad m+1]$ is presented in this section. The moduli $m_i = [m-1 \quad m \quad m+1]$ can also be written in the more familiar form $m_i = [2^r - 1 \quad 2^r \quad 2^r + 1]$ with the restriction that 2^r must be an even integer. This does not mean the variable *r* has to be an integer or even a rational number, as will be seen shortly. The longest sequence of unique RSNS vectors for this particular class of moduli has an upper bound that is determined by particular Case 312 redundancy and a lower bound that is determined by a particular Case 220 redundancy. This can be proven by calculating all of the redundancies for all the cases listed in Figure 4.16, ordering all redundancies consecutively in the fundamental period, and then calculating which redundancy vectors bound the largest series of non-redundant RSNS vectors. Expressions describing the length and position of the redundancies in Case 312 and Case 220 have already been provided in Figure 4.16. Therefore, it is possible to form an analytic solution for $\hat{M}$ for this particular class of moduli.

1. $\hat{M}$ Upper Bound

The longest sequence of unique RSNS vectors for $m_i = [m-1 \quad m \quad m+1]$ is bounded by the Case 31X redundancy with the largest COR. This corresponds to Case 312. The Case 312 system of congruence equations from (4.25) is

$$\begin{aligned}
\frac{k}{6} &\equiv -\frac{(h-2)}{3} \pmod{m_1}, \\
\frac{k}{6} &\equiv -\frac{(h+1)}{3} \pmod{m_2}, \\
\frac{k}{6} &\equiv -\frac{(h+1)}{3} \pmod{m_3},
\end{aligned} \tag{4.81}$$

and have the solution

$$h_{Case312} = \left(a(3m_1m_2m_3) + h_{shift312}\right) - \frac{k}{2}, \tag{4.82}$$

where $h_{shift312}$ is found from the LPS to

$$\begin{aligned}
\frac{\left(h_{shift312} - 2\right)}{3} &\equiv 0 \pmod{m_1}, \\
\frac{\left(h_{shift312} + 1\right)}{3} &\equiv 0 \pmod{m_2}, \\
\frac{\left(h_{shift312} + 1\right)}{3} &\equiv 0 \pmod{m_3}.
\end{aligned} \tag{4.83}$$

For this case, *k* is a multiple of six. Rearranging and simplifying, the system of equations in (4.83) become

$$\begin{aligned}
x_0 &\equiv 1 \pmod{m_1}, \\
x_0 &\equiv 0 \pmod{m_2}, \\
x_0 &\equiv 0 \pmod{m_3},
\end{aligned} \tag{4.84}$$

where

$$x_0 = \frac{\left(h_{shift312} + 1\right)}{3}. \tag{4.85}$$

The system of equations in (4.84) can be solved using the CRT. The well-known solution to the CRT is

$$x_0 = \sum_i \frac{M}{m_i} a_i b_i, \tag{4.86}$$

where a_i are the residues and b_i can be found by the Euclidian Algorithm [28]. Since only the residue a_1 is non-zero in (4.84), equation (4.86) reduces to

$$x_0 = m_2 m_3 a_1 b_1 . \tag{4.87}$$

According to the Euclidian Algorithm, the coefficient b_i is the factor that solves

$$\left(\frac{M}{m_i}\right) b_i + \left(m_i\right) c_i = 1, \tag{4.88}$$

where b_i and c_i are any integers. The table in Figure 4.20 shows results of repeatedly solving (4.88) using the Euclidian Algorithm with moduli $m_i = [2^r - 1 \;\; 2^r \;\; 2^r + 1]$ for selected values of r.

r	$m_1 = 2^r$ - 1	$m_2 = 2^r$	$m_3 = 2^r + 1$	b_1	b_2	b_3
2	3	4	5	2	3	3
$\log_2 6$	5	6	7	3	5	4
3	7	8	9	4	7	5
$\log_2 10$	9	10	11	5	9	6
$\log_2 12$	11	12	13	6	11	7
$\log_2 14$	13	14	15	7	13	8
4	15	16	17	8	15	9
5	31	32	33	16	31	17

Figure 4.20 Tabular solutions to the CRT using the Euclidian Algorithm.

By inspection of Figure 4.20, it is apparent that for moduli with the form $m_i = [2^r - 1 \;\; 2^r \;\; 2^r + 1]$ the b_i coefficients have the form

$$\begin{aligned} b_1 &= 2^{r-1}, \\ b_2 &= 2^r - 1, \\ b_3 &= 2^{r-1} + 1. \end{aligned} \tag{4.89}$$

Note that nowhere is r required to be an integer or even a rational number. Only $b_1 = 2^{r-1}$ is required to be an integer, which means that 2^r must be an even integer. The validity of the coefficients can be proved by substitution into (4.88). For the coefficient b_1, the proof is

$$
\begin{aligned}
(m_2 m_3)(2^{r-1}) + (m_1)(-2^{2r-1} - 2^r - 1) &= 1 \\
(2^r)(2^r + 1)(2^{r-1}) + (2^r - 1)(-2^{2r-1} - 2^r - 1) &= 1 \\
(2^{3r-1} + 2^{2r-1}) + (-2^{3r-1} - 2^{2r} + 2^{2r-1} + 1) &= 1 \\
(2^{2r-1} + 2^{2r-1}) - 2^{2r} + 1 &= 1 \\
2^{2r} - 2^{2r} + 1 &= 1 \\
1 &= 1.
\end{aligned}
\tag{4.90}
$$

For the coefficient b_2, the proof is

$$
\begin{aligned}
(m_1 m_3)(2^r - 1) + (m_2)(-2^{2r} + 2^r + 1) &= 1 \\
(2^r - 1)(2^r + 1)(2^r - 1) + (2^r)(-2^{2r} + 2^r + 1) &= 1 \\
(2^{3r} - 2^{2r} - 2^r + 1) - (2^{3r} - 2^{2r} - 2^r) &= 1 \\
1 &= 1.
\end{aligned}
\tag{4.91}
$$

For the coefficient b_3, the proof is

$$
\begin{aligned}
(m_1 m_2)(2^{r-1} + 1) + (m_3)(1 - 2^{2r-1}) &= 1 \\
(2^r - 1)(2^r)(2^{r-1} + 1) + (2^r + 1)(1 - 2^{2r-1}) &= 1 \\
(2^{3r-1} + 2^{2r} - 2^{2r-1} - 2^r) + (-2^{3r-1} + 2^r - 2^{2r-1} + 1) &= 1 \\
-(2^{2r-1} + 2^{2r-1}) + 2^{2r} + 1 &= 1 \\
-2^{2r} + 2^{2r} + 1 &= 1 \\
1 &= 1.
\end{aligned}
\tag{4.92}
$$

Although only b_1 is of interest for the calculation of the Case 312 COR, the analytical solution for b_2 and b_3 will be essential in subsequent sections. Andraos and Ahmad [37] calculated similar results for the analytical expressions of the coefficients using a different method of analysis and limiting r to integer values.

Returning to Case 312, the solution to the general form of the CRT for the Case 312 COR from (4.85) and (4.87) is

$$
\frac{\left(h_{shift312} + 1\right)}{3} = x_0 = m_2 m_3 a_1 b_1 . \tag{4.93}
$$

All quantities are known and the solution can be expressed in terms of the parameter r as

$$\frac{\left(h_{shift312}+1\right)}{3}=(2^r)\left(2^r+1\right)(1)\left(2^{r-1}\right)=2^{3r-1}+2^{2r-1}. \tag{4.94}$$

Rearranging, the solution becomes

$$h_{shift312}=3\left(2^{3r-1}+2^{2r-1}\right)-1. \tag{4.95}$$

Since by definition, $\hat{M}$ cannot contain both redundant vectors in the Case 312 redundancy, the upper bound for $\hat{M}$ must be one less than the upper bound of the Case 312 redundancy. Therefore, substituting (4.95) into the solution for the Case 312 system of congruence equations (4.82) and subtracting one yields

$$\begin{aligned} UB &= h_{shift312}+\frac{k}{2}-1 \\ &= 3\left(2^{3r-1}+2^{2r-1}\right)-1+3-1 \\ &= 3\left(2^{3r-1}+2^{2r-1}\right)+1, \end{aligned} \tag{4.96}$$

using the least positive value for *k* and the least positive COR. This is the upper bound for $\hat{M}$ for moduli of the form $m_i=[m-1 \;\; m \;\; m+1]$.

2. $\hat{M}$ Lower Bound and Length

The lower bound for $\hat{M}$ for the moduli $m_i=[m-1 \;\; m \;\; m+1]$ is constrained by the lower bound of a particular Case 220 redundancy. The congruence equations for this case are provided in (4.41) and have the solution of

$$\begin{aligned} h_{Case220} &= -3m_2+a(3m_1m_3) \\ &= -3(2^r)+a3(2^r-1)(2^r+1), \end{aligned} \tag{4.97}$$

using the least positive *k*. Unfortunately, the choice for the parameter *a* is not straightforward. Multiples of the Case 220 COR start at zero and repeat every $3m_1m_2$. Since the fundamental period of the RSNS is $6m_1m_2m_3$, there are exactly $2m_3$ Case 220 COR in the fundamental period. The Case 220 COR of interest, $a(3m_1m_3)$, is the largest one that

is less than or equal to the Case 312 COR (4.95) used to calculate the $\hat{M}$ upper bound. Substituting $(2^r - 1)$ and $(2^r + 1)$ for m_1 and m_3, the resulting inequality and solution is

$$\begin{aligned} 3\left(2^{3r-1} + 2^{2r-1}\right) - 1 &\geq a3(2^r - 1)(2^r + 1) \\ 2^{3r-1} + 2^{2r-1} &\geq a(2^{2r} - 1) \\ \frac{2^{3r-1} + 2^{2r-1}}{(2^{2r} - 1)} &\geq a \\ 2^{r-1} + 2^{-1} - 2^{-r-1} + \cdots &\geq a. \end{aligned} \tag{4.98}$$

Since a must be an integer and for $r > 1$ the sum of all the factors in the last line of (4.98) to the right of 2^{r-1} is a positive fraction, then the choice for a must be 2^{r-1}. Since by definition, $\hat{M}$ cannot contain both redundant vectors in the Case 220 redundancy, the lower bound for $\hat{M}$ must be one more than the lower bound of the Case 220 redundancy. Using this result and the lower bound of the Case 220 redundancy from (4.97), the lower bound of $\hat{M}$ is

$$\begin{aligned} LB &= -3m_2 + a(3m_1m_3) + 1 \\ &= -3(2^r) + (2^{r-1})3(2^r - 1)(2^r + 1) + 1 \\ &= 3(2^{3r-1} - 2^{r-1} - 2^r) + 1. \end{aligned} \tag{4.99}$$

Therefore, the size of $\hat{M}$ is given by

$$\begin{aligned} \hat{M} &= UB - LB + 1 \\ &= 3\left(2^{3r-1} + 2^{2r-1}\right) + 1 - 3(2^{3r-1} - 2^{r-1} - 2^r) - 1 + 1 \\ &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1. \end{aligned} \tag{4.100}$$

Pace, Styer, and Akin [30] conjecture that, based on repeated example, $\hat{M}$ for moduli $m_i = [2^r - 1 \quad 2^r \quad 2^r + 1]$ with r an integer is given by

$$\frac{3}{2}m_1^2 + \frac{15}{2}m_1 + 7. \tag{4.101}$$

The following proof demonstrates that (4.101) agrees with the results of (4.100) presented in this paper.

$$
\begin{aligned}
\frac{3}{2}m_1^2 + \frac{15}{2}m_1 + 7 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1 \\
3(2^{-1})(2^r - 1)^2 + 15(2^{-1})(2^r - 1) + 7 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1 \\
3(2^{2r-1} - 2^r) + 15(2^{r-1}) + 1 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1 \\
3(2^{2r-1}) - 3(2^r) + 3(2^{r-1}) + 12(2^{r-1}) + 1 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1 \\
3(2^{2r-1}) + 3(2^{r-1}) + 6(2^r) - 3(2^r) + 1 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1 \\
3(2^{2r-1}) + 3(2^{r-1}) + 3(2^r) + 1 &= 3(2^{2r-1} + 2^{r-1} + 2^r) + 1.
\end{aligned}
\tag{4.102}
$$

The benefit of the research presented here over the previous work by Pace, Styer, and Akin is that (4.100) provides both length and location of $\hat{M}$, while the previous work produced conjectured length calculations only. Furthermore, this research extends the solution to all PRP moduli of the form $m_i = [m-1 \;\; m \;\; m+1]$.

It is very likely that more special classes of moduli exist for the RSNS *N*-modulus case that have closed-form analytical expressions for $\hat{M}$. However, the need still exists for an efficient search algorithm to find $\hat{M}$ for an arbitrary set of *N* moduli.

E. *N*-MODULUS RSNS $\hat{M}$ SEARCH ALGORITHM

1. RSNS Redundancy Vector Graphical Representation

The expressions for the *N*-modulus RSNS redundancies developed in the previous sections provide a complete mathematical description of the RSNS. However, in their current form, the redundancy expressions do not provide any insight into how they may be used to compute the location and size of $\hat{M}$ for an arbitrary set of PRP moduli. This section describes a novel RSNS graphical display that provides an intuitive presentation of the vector redundancy equations. The inspiration for the new display springs from an unlikely source, a geographic information system (GIS).

a. Geographical Information Systems

GIS is a method to visualize, manipulate, analyze, and display spatial data [38]. GIS is computer software that combines geographic information (where things are) with descriptive information (what things are) in a single display and analysis tool. Unlike a flat paper display, a GIS can simultaneously present many selectable layers of information as shown in Figure 4.21.

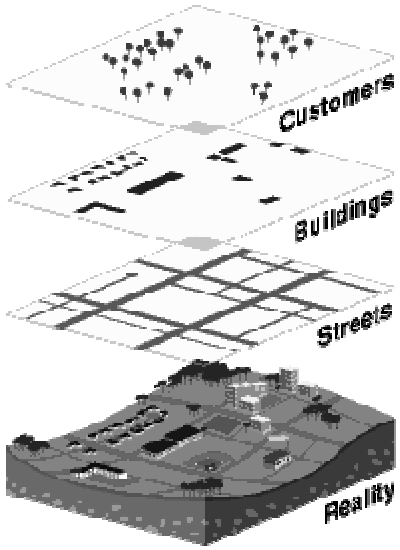

Figure 4.21 An example of selectable data layers in a GIS (From [40].)

The geographic data can be thought of as layers of information underneath the computer screen. Each layer represents a particular feature of the display. For example, on a digital map, one layer could be made up of all the streets in an area. Another layer could represent all the buildings in the same area. Yet another could represent the customers of a particular business. These layers can be laid on top of one another, creating a stack of information about the same geographic area. Each layer can be selected or deselected individually, like peeling a layer off a stack of transparencies. Furthermore, each "smart map" layer has an associated database such that each feature in a layer is linked to information related to the feature. For instance, the database entry associated with a building on a layer may have information about the purpose, age, or square footage of the particular building. In a GIS, the user controls the amount of information displayed about any area desired, at any time, on any specific map [39].

GIS is often called "mapping software." Maps are usually associated with physical geography, but Figure 4.22 demonstrates that GIS is flexible enough to map any kind of terrain, even the human body.

Figure 4.22 Human body “geography” (From [38].).

It is estimated that 80% of all data, not just geographic, has a spatial component [38]. Surprisingly, this includes the robust symmetric number system. Consequently, GIS software can be used to analyze this particular number system by displaying user-selected information symbolized to highlight specific characteristics. Moreover, pattern recognition is something at which human beings excel. There is a vast difference between seeing data in a table of rows and columns and seeing it presented in the form of a map. The difference is not simply aesthetic, it is conceptual - it turns out the way data is seen has a profound effect on the connections made and the conclusions drawn from it. GIS provides the layout and drawing tools that help present facts with clear, compelling visualizations [40]. The next section presents a unique display of the RSNS vectors and redundancies, which allows the viewer to apply innate pattern recognition skills to gain insight into the nature of the RSNS.

b. RSNS Circle Representation

Since the RSNS is periodic, it is permissible to represent the number system as a circle that, when plotted using a GIS, provides a more intuitive display than a linear representation. The circle has a fixed radius and a finite width and can be considered a "map." This enables the principles of geometry to be leveraged against the problem of locating $\hat{M}$. Furthermore, a GIS permits data to be associated with each element on the plot, allowing a user to drill down into the RSNS display for linked non-graphical information. Figure 4.23 shows one such circular RSNS representation where each block in the circle represents an RSNS vector starting at $h = 0$ at the three o'clock position on the circle and increasing counterclockwise.

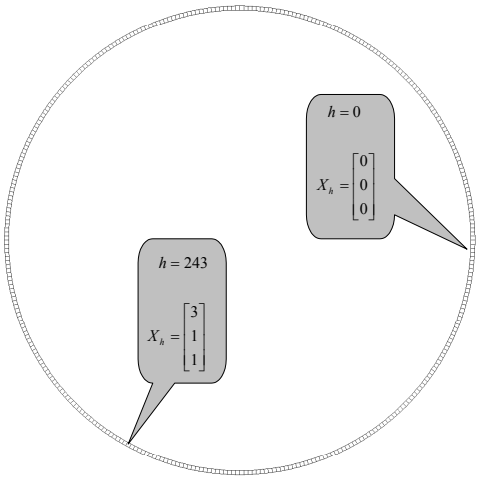

Figure 4.23 Circular representation of the RSNS.

The example in the figure and all other examples in this section is a three-modulus RSNS with $m_i = [3 \quad 4 \quad 5]$ unless otherwise noted. The GIS software used for the display and analysis of the RSNS circle plot was the ArcGIS suite produced by ESRI. ArcGIS contains visualization and analysis features that enable the RSNS to be studied in greater detail than before. ArcGIS provides selectable layers with associated database tables, transparent layers to enable simultaneous multiple layer visualizations, and built-in analytical functions for querying or sorting by geographic features and database fields.

Furthermore, the ArcGIS software is extendable by using common programming languages (e.g., Visual Basic) to create additional custom GIS functionality. MATLAB was used to create text files of the RSNS vectors, redundancy locations, and $\hat{M}$ locations. The text files were converted to ArcGIS format using a custom Microsoft Visual Basic program that created the various data layers, called "shapefiles," and the associated database tables. The fully-commented MATLAB and Visual Basic programs used to create the shapefiles are in Appendix C.

To understand the relationships between the various case vector redundancies, it is useful to think of the RSNS vectors and redundancies as areas. To facilitate this concept, arcs whose endpoints correspond to the redundant vectors represent the vector redundancies in the circle plot. An example of the RSNS circle representation showing the Case 210, Case 220, and Case 230 redundancy arcs is provided in Figure 4.24.

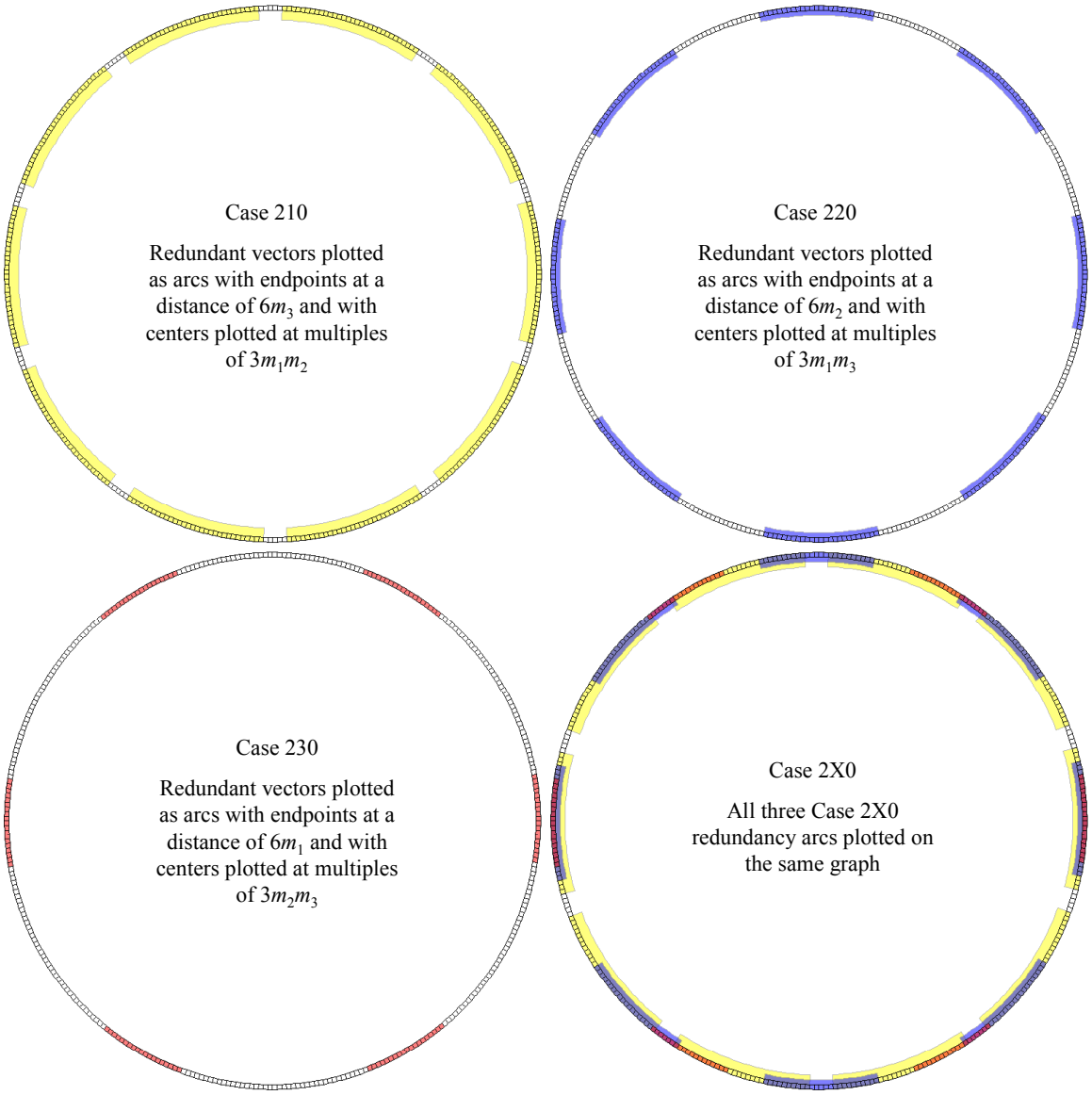

Figure 4.24 Circular RSNS representation with Case 2X0 redundancy arcs.

The redundancy arcs in the figure were plotted using the expressions for the vector redundancies from the table in Figure 4.16 using the least positive k. Using larger k would result in larger symmetric arcs centered at the same COR. Thus, if the endpoints of the arc using the larger k restricted the size of $\hat{M}$, the redundancy arc with the smaller k would restrict the size of $\hat{M}$ even more. Therefore, it was only necessary to consider the redundancy arcs plotted using the least positive k value.

Analysis of the RSNS circle representation using ArcGIS provided several insights regarding the nature of the RSNS. The list below details a few of the more important findings with regard to the RSNS redundancy locations.

- The largest arc of the circle that does not completely overlap a redundancy arc is $\hat{M}$.
- Only redundancy arcs computed using least positive *k* affect $\hat{M}$.
- All redundancy arcs have an equal-sized redundancy arc (same size and case only, not necessarily composed of the same vectors) on the opposite side of the circle (i.e., the COR are separated by exactly $P_f / 2$)
- The 0^{th} MRSS redundancy arcs are symmetric about the vertical and horizontal bisectors (diameters) of the circle.
- All MRSS redundancies for any case can be found by computing the 0^{th} MRSS redundancy and shifting each COR by an amount equal to the shift computed for the corresponding Case *N*1X MRSS.

The first point stems from the fact that, by definition, $\hat{M}$ contains no ambiguities and therefore no redundant vectors. Since the redundancy arcs represent redundant vectors, the $\hat{M}$ arc cannot fully overlap a redundancy arc (but partial overlap is permitted). The second point was discussed in the previous paragraph. The third point can be proven by the following argument. Given a COR for any case at

$$h = a\left(N\prod_i m_i \right), \tag{4.103}$$

where the subscript *i* are the indices of all MRSs with Type 1 redundancies, there is also a COR at $h + \left(P_f / 2 \right)$ because

$$\begin{aligned} a\left(N\prod_i m_i \right) + \frac{P_f}{2} &= a\left(N\prod_i m_i \right) + N\prod_{n=1}^{N} m_n \\ &= a\left(N\prod_i m_i \right) + N\left(\prod_i m_i \prod_j m_j \right) \\ &= \left(a + \prod_j (m_j) \right)\left(N\prod_i (m_i) \right) \\ &= b\left(N\prod_i m_i \right), \end{aligned} \tag{4.104}$$

where a and b are any integers, j correspond to the MRSs with Type 0 redundancies, and i correspond to the MRSs with Type 1 redundancies.

The fourth point, although easy to see on the circle plot in Figure 4.24, is more difficult to prove. First, the expressions in the table in Figure 4.19 show that there are always an even number of COR in each case. This is because given the COR in (4.103), the number of COR in a fundamental period is

$$\begin{aligned} \frac{P_f}{N\prod_i m_i} &= \frac{2N\prod_{n=1}^{N} m_n}{N\prod_i m_i} \\ &= 2\prod_j m_j . \end{aligned} \tag{4.105}$$

Since the number of redundancies is even and there is always a 0^{th} MRSS redundancy at $h = 0$, geometry says there must always also be a 0^{th} MRSS redundancy at $h = P_f / 2$. Geometry also mandates that, since the COR are evenly spaced throughout the circle and there are COR centered at $h = 0$ and $h = P_f / 2$, there must be an equal number of evenly spaced COR above and below the horizontal diameter of the circle. Similarly, since there is an equal number of evenly spaced COR above and below the horizontal diameter of the circle, they must be symmetric on either side of the vertical diameter of the circle.

The last point in the list above is a conjecture based on pattern recognition in the structure of the RSNS circle plot, which has proven correct through the thousands of moduli sets tested thus far. The thrust of the conjecture is that the 0^{th} MRSS redundancies and the COR shifts computed in Case N1X provide necessary and sufficient conditions to completely describe all redundancies for any N-modulus set of PRP moduli.

The consequences of the points above lead directly to the following conclusions, which are used to formulate an efficient RSNS N-modulus $\hat{M}$ search algorithm:

- Every $\hat{M}$ has an identically sized $\hat{M}$ composed of different vectors on the exact opposite side of the circle (i.e., the corresponding points in each arc are separated by $P_f / 2$).

- There is always at least one $\hat{M}$ in the upper-half of the circle and in the lower half of the circle so it is only necessary to consider the vectors from $h = -N$ to $h = \left(P_f / 2\right) + N$ when searching for $\hat{M}$.
- When computing the 0^{th} MRSS redundancy arcs, only the arcs in the first quadrant need to be computed and the rest of the quadrants can be found by reflection.
- The redundancy arcs for all MRSSs other than the 0^{th} MRSS can be computed by shifting the 0^{th} MRSS arcs.
- The redundancy arcs limit the length of $\hat{M}$, so the case with the smallest distance between the leading and trailing edges of consecutive redundancy arcs provides an upper bound on the length of $\hat{M}$.
- Any redundancy arc that is larger than $\hat{M}$ length upper bound does not affect the size of $\hat{M}$ and can be ignored.
- Once all redundancies smaller than the $\hat{M}$ length upper bound have been computed and combined, $\hat{M}$ is found where the largest sequence of vectors exists between the leading and trailing edges of consecutive redundancy arcs across the different cases.

2. *SmartSearch* $\hat{M}$ Search Algorithm

A new $\hat{M}$ search algorithm based on the conclusions detailed in the previous section was created and implemented using MATLAB. The name given to the algorithm is *SmartSearch*. The fully-commented MATLAB programs realizing the *SmartSearch* algorithm are in Appendix C. *SmartSearch* drastically reduces the time required to find the $\hat{M}$ size and location by more than five orders of magnitude compared to an unpublished search program by Pace and Styer. Furthermore, since the search algorithm in this research was based on a *circular* RSNS representation, it correctly identifies those unique sequences of RSNS vectors that wrap around at the fundamental period boundary. The Pace/Styer search program was used to compute the data presented in [30] and is also provided for reference in Appendix C. Figure 4.25 shows a plot of $\hat{M}$ versus computation time for the two search programs using hundreds of *N*-modulus moduli sets.

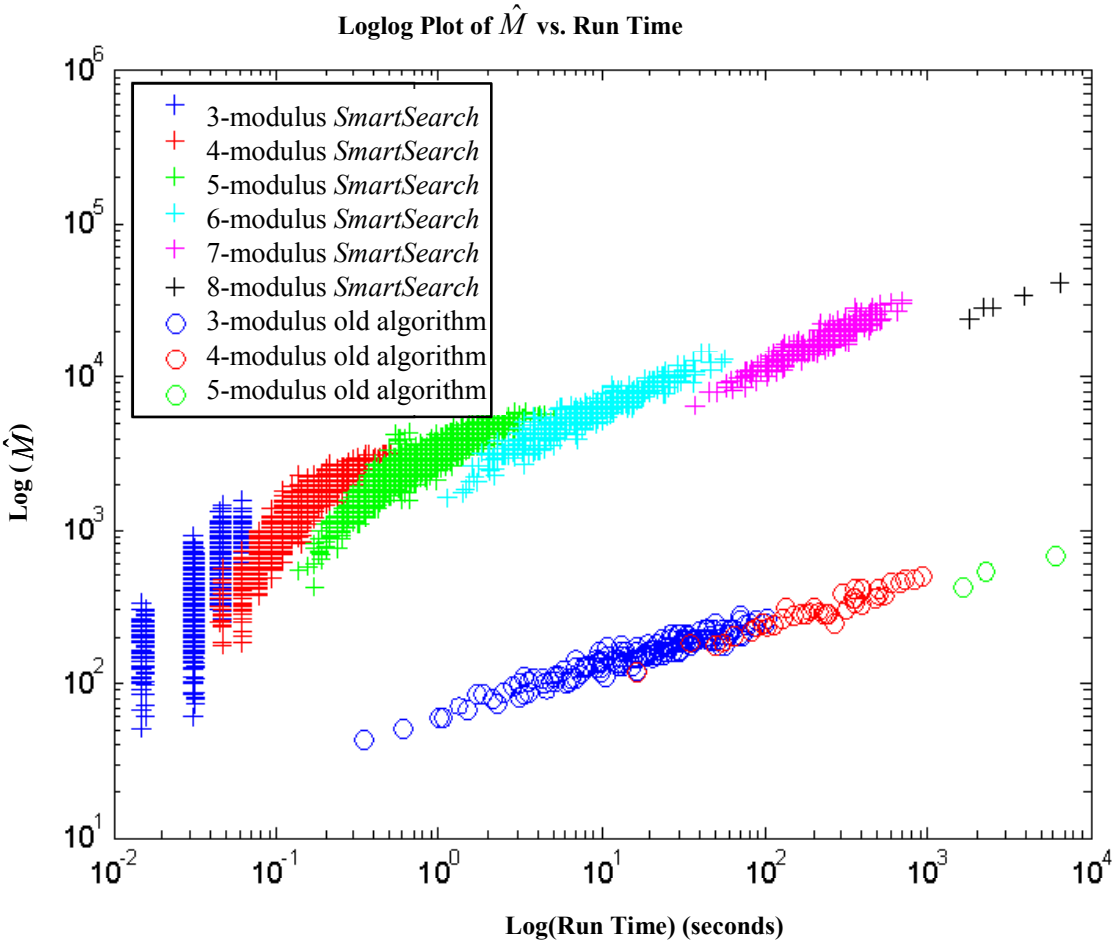

Figure 4.25 Plot of $\hat{M}$ vs. run time using new and old search algorithms.

The figure illustrates the efficiency of the search algorithm developed in this research, which is based on computing redundancies, compared to the previous search algorithm, which is based on searching the vector strings for $\hat{M}$. From the figure, it is evident that the *SmartSearch* algorithm is as much as five orders of magnitude faster than the old search algorithm, and that the advantage of *SmartSearch* increases with the length of $\hat{M}$. In fact, based on linear interpolation of the data in the figure, it would take the Pace/Styer program more than 32 years to find $\hat{M}$ for the same 8-modulus RSNS realization that the *SmartSearch* algorithm computed in less than 2 hours. Moreover, the *SmartSearch* algorithm uses far less memory than the Pace/Styer algorithm and therefore can find the *N*-modulus $\hat{M}$ for moduli sets with much larger fundamental periods. For

example, the *SmartSearch* algorithm successfully computed $\hat{M}$ for an eight-modulus system with a fundamental period of close to *one billion*, while the Pace/Styer algorithm could not successfully compute an RSNS moduli set with a fundamental period of only *three million* due to memory limitations. Therefore, since the *SmartSearch* algorithm was uniquely able to compute $\hat{M}$ for thousands of *N*-modulus RSNS moduli sets in a relatively short time, it was possible to find $\hat{M}$ for many moduli sets and to study the advantages *N*-modulus RSNS. For instance, Figure 4.26 shows a plot of $\hat{M}$ versus moduli sum $\left(\sum_i m_i\right)$.

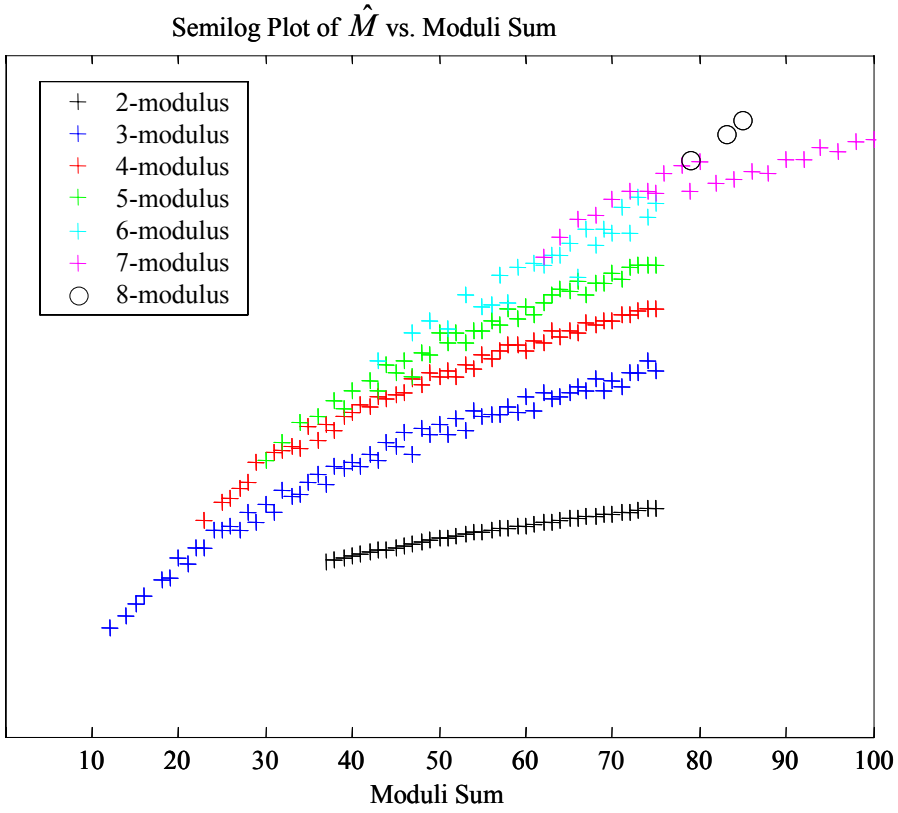

Figure 4.26 Plot of $\hat{M}$ vs. moduli sum for two-modulus to eight-modulus systems.

From the figure, it is evident that for any moduli sum, the largest $\hat{M}$ occurs in the RSNS with the largest number of moduli. In some cases, an increase in $\hat{M}$ greater than two orders of magnitude was accomplished simply by regrouping the same moduli sum into a larger number of moduli. Put in terms of hardware design of an ADC using the

RSNS, this translates to an increase of two orders of magnitude in $\hat{M}$ with little if any increase in hardware.

Most practical *r*-bit ADCs are designed so that their dynamic ranges are on 2^r boundaries. Therefore, the table in Figure 4.27 was generated using the *SmartSearch* program and lists all two-modulus through eight-modulus minimum-sum moduli sets that have an $\hat{M}$ greater or equal to *k*-bits.

Sum	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	P_f	Start	Stop	$\hat{M}$	r-bit $\hat{M}$	2^r
13	6	7							168	17	49	33	5	32
12	3	4	5						360	79	121	43	5	32
24	11	13							572	5	69	65	6	64
16	4	5	7						840	-2	63	66	6	64
45	22	23							2024	49	177	129	7	128
24	7	8	9						3024	733	865	133	7	128
23	3	5	7	8					6720	2070	2250	181	7	128
32	9	10	13						7020	-2	273	276	8	256
29	5	7	8	9					20160	8060	8466	407	8	256
30	3	4	5	7	11				46200	2336	2755	420	8	256
44	11	16	17						17952	6815	7340	526	9	512
35	7	8	9	11					44352	1361	2010	650	9	512
34	3	5	7	8	11				92400	23247	23934	688	9	512
62	18	19	25						51300	-2	1029	1032	10	1024
47	9	11	13	14					144144	67580	68825	1246	10	1024
40	5	7	8	9	11				277200	56908	57984	1077	10	1024
43	3	4	5	7	11	13			720720	218663	220276	1614	10	1024
61	13	15	16	17					424320	193099	195172	2074	11	2048
50	7	9	10	11	13				900900	188310	190625	2316	11	2048
47	3	5	7	8	11	13			1441440	673896	676217	2322	11	2048
64	7	11	13	16	17				2722720	450759	455021	4263	12	4096
57	5	7	8	9	11	17			5654880	802784	807881	5098	12	4096
62	3	4	5	7	11	13	19		15975960	7100574	7107079	6506	12	4096
67	7	9	10	11	13	17			18378360	7335188	7344640	9453	13	8192
64	3	5	7	8	11	13	17		28588560	12345388	12354000	8613	13	8192
76	5	7	8	9	11	17	19		125349840	53253852	53273736	19885	14	16384
79	3	4	5	7	11	13	17	19	310390080	47189369	47213577	24209	14	16384
83	3	5	7	8	11	13	17	19	620780160	48265355	48299734	34380	15	32768

Figure 4.27 Table of minimum-sum moduli sets with $\hat{M}$ larger than *r*-bits for $5 \le r \le 15$.

For example, using the figure above, an ADC with a 10-bit $\hat{M}$ can be designed with a 3-modulus, 4-modulus, 5-modulus, or 6-modulus RSNS using the moduli sets provided above. Moreover, the 7-modulus or 8-modulus RSNS provided above would also

suffice, but they would require more hardware while providing no additional benefit. There is also a 2-modulus RSNS that has a 10-bit $\hat{M}$, but the moduli would be very large relative to the moduli sets presented above.

F. SUMMARY

The Gray-code property of the RSNS makes it particularly useful in folding analog-to-digital converters (ADC), direction finding interferometer antenna architectures, and electro-optic digital antennas since it eliminates encoding errors common in these systems. Although [33] provided an expression for the *length* of $\hat{M}$ for a two-modulus RSNS, inefficient computer search algorithms had to be used to find the $\hat{M}$ *position* for the two-modulus case as well as for $\hat{M}$ for all RSNS with greater than two moduli. This chapter presented detailed mathematical descriptions of the three-modulus RSNS, as well as an analytic expression for $\hat{M}$ length and position for a popular set of moduli $m-1,\ m,\ m+1$ for *m* even. Furthermore, the three-modulus results were extended into a detailed mathematical description of the *N*-modulus RSNS. Although the mathematical description of the *N*-modulus redundancies was complete, the expressions themselves were not enough to provide insight into the nature of the RSNS. Therefore, a novel geographic-based RSNS circle representation was developed to provide a format more suitable to visual analysis and pattern recognition. As a result, enough exploitable properties of the RSNS were discovered and enabled the development of an efficient $\hat{M}$ search algorithm, *SmartSearch*. The speed and memory efficiency of SmartSearch is several orders of magnitude above the only other known $\hat{M}$ search algorithm. Sample results from the *SmartSearch* algorithm were presented that provided efficient RSNS moduli sets for the design of 5- to 15-bit ADCs.

The next chapter builds on the RSNS theory presented herein to develop a procedure for RSNS-to-binary conversion. The RSNS-to-binary conversion procedure is used to construct digital circuits capable of converting the thermometer-coded output of the ADCs from Chapter III to binary.

V. RSNS-TO-BINARY CONVERSION

Chapter IV and [41] present essential theory and methods required to compute an entire fundamental period of RSNS residue vectors and determine the longest sequence of unique vectors ($\hat{M}$) given a set of pair-wise relatively prime moduli. Chapter III provides the design of an RSNS folding ADC that produces a thermometer-coded representation of an RSNS residue vector. However, one of the fundamental difficulties in a hardware application of the RSNS, such as an analog-to-digital converter, is the conversion of the thermometer-coded RSNS residue vector to the corresponding position of the residue vector in the fundamental period of the RSNS. Furthermore, since it is very likely that the residue vector occurs at multiple positions in the RSNS fundamental period, it is necessary to determine which position occurs within $\hat{M}$ [42]. In a residue number system (RNS), the residue vector can be converted to a position in the fundamental period by forming a single system of congruence equations that can be solved using the CRT. Furthermore, each residue vector occurs only once in the RNS fundamental period. By contrast, the RSNS symmetrical residues cannot be converted to a position in the fundamental directly using the CRT since the integers within each RSNS modulus residue sequence are ambiguous. This chapter presents three methods of converting the thermometer-coded RSNS residue vectors to a binary representation of the residue vector position within $\hat{M}$. The first two methods are straightforward schemes, which convert the thermometer code to binary using a ROM and a decoder. The third method is an algorithmic approach that takes advantage of an underlying RNS structure present in RSNS.

A. RSNS-TO-BINARY CONVERSION USING CONVENTIONAL TECHNIQUES

1. ROM Conversion

A read only memory (ROM) is conceptually the simplest hardware means of converting the RSNS residues to binary. Furthermore, it is the only method mentioned in published research that converts the RSNS residues to binary. Unfortunately, it is also the slowest technique and requires more die area than any of the alternative approaches proposed in this dissertation. Moreover, a ROM decoding scheme does not lend itself easily to pipelining. To facilitate the size comparison between the three decoding schemes, an RSNS with moduli

$$M_{345} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}, \tag{5.1}$$

is used as an example throughout this chapter. The RSNS ROM decoding method involves decoding the 12 comparator outputs that drive 2^{12} address lines in a 2^{12} by 6-bit ROM. A more efficient ROM implementation is one that converts the thermometer bits in each channel to a Gray code using the algorithm in [28], thereby reducing the 12 comparator output bits to 8 bits. As a result, an 8 to 2^8 address line decoder drives the inputs to a 2^8 by 6-bit ROM. In round numbers, the first ROM design requires approximately 129,000 transistors while the second design requires only 6000 transistors. These are conservative estimates that include only the transistors necessary to complete the required logic functions. No additional circuitry, such as buffers, are included in the estimates.

2. Decoder Conversion

The second method to convert the RSNS thermometer code to binary is a decoding scheme. The thermometer code is converted to Gray code like the second ROM design in the previous section. The resulting 8 bits and their complements are used to form 43 product terms corresponding to the 43 unique residue vectors in the $\hat{M}$ produced by

the RSNS with the moduli in (5.1). Finally, the product terms are encoded to form the six-bit binary output. Figure 5.1 shows the structure of the decoder with the Gray code encoder block and the eight product term blocks. Each product term block produces six product terms except the final block, which produces only one product term.

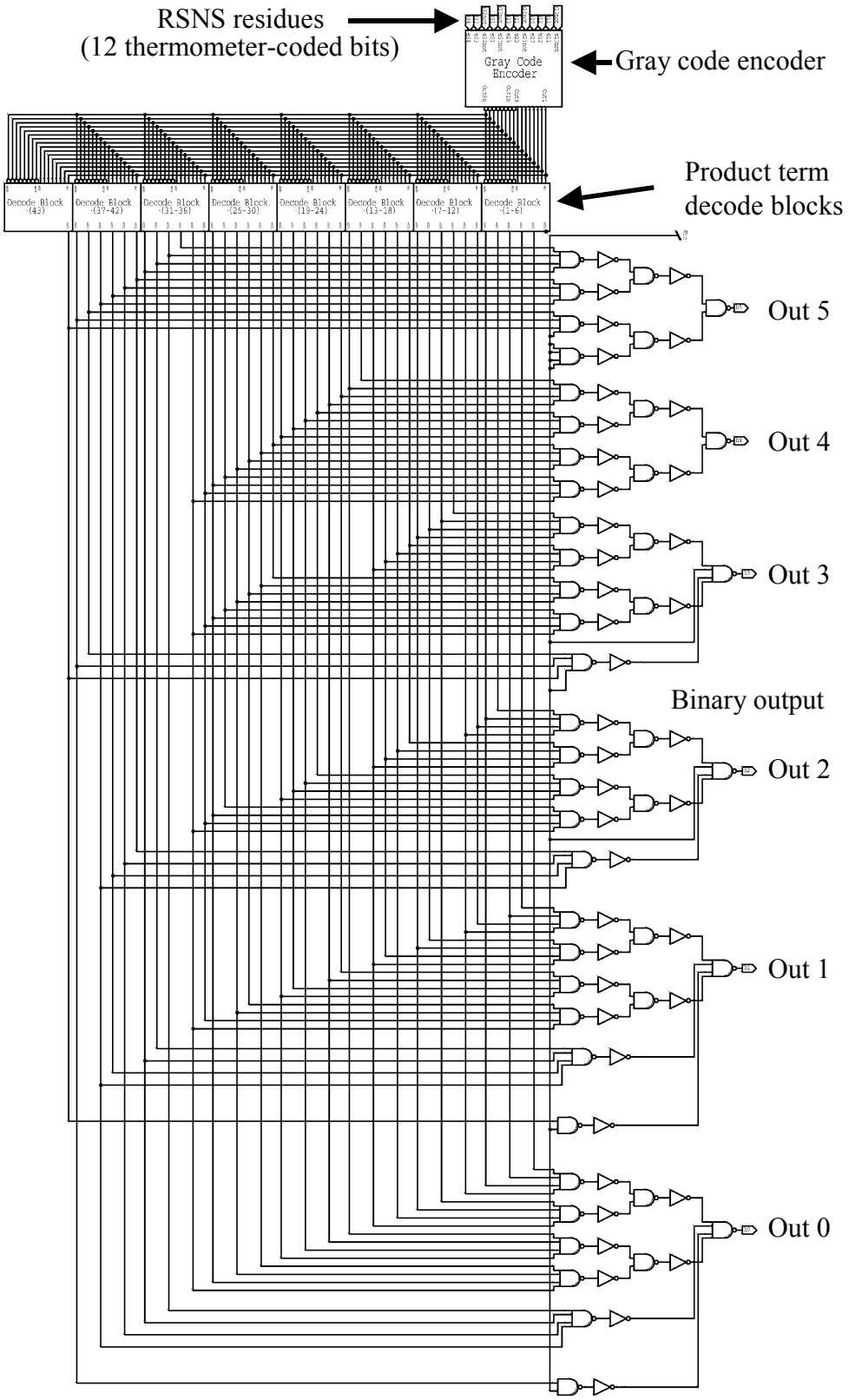

Figure 5.1 RSNS to binary conversion using a decoding circuit.

Figure 5.2 shows the structure of the thermometer-to-Gray-code conversion as described in [28], and Figure 5.3 shows the first decode block that generates the first six product terms.

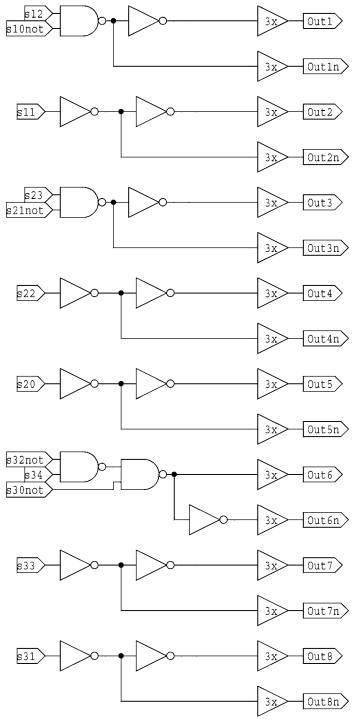

Figure 5.2 Thermometer-code to Gray-code conversion.

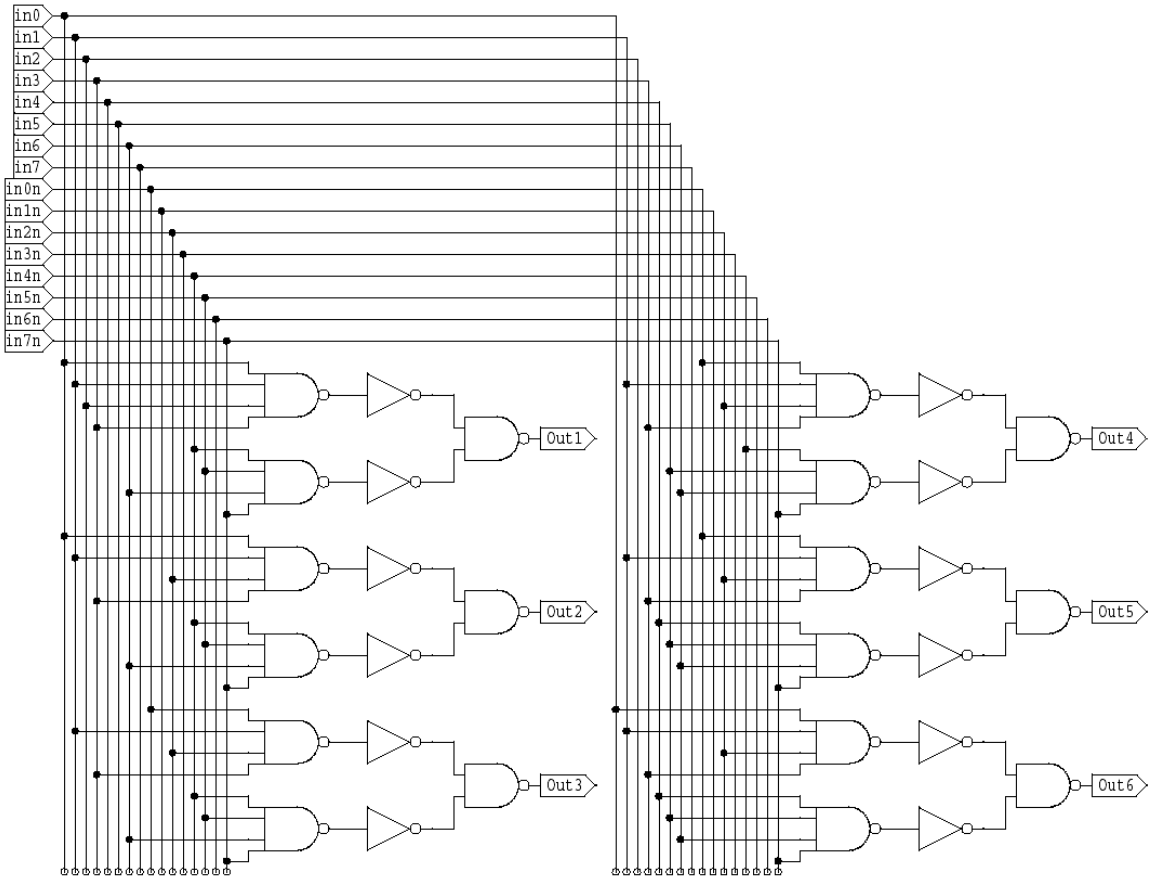

Figure 5.3 Decode block for the first six RSNS vector positions.

The total size of the decoder is approximately 1700 transistors. Other than the size advantage, the benefit of this conversion method over the ROM design is that it can be pipelined. The drawback to the ROM and decode methods is that as the number of ADC thermometer output bits (or comparators) increases, the number of transistors used to form the decoding schemes increases exponentially. Furthermore, the number of pipeline registers needed to pipeline the decoder scheme is equal to $\hat{M}$. Thus, both simple schemes are extremely inefficient for ADCs much larger than six bits. A more efficient algorithmic approach to converting RSNS to binary that exploits the underlying structure of the RSNS is presented in the next section.

B. THREE-MODULUS RSNS-TO-BINARY CONVERSION

This section develops an RSNS-to-binary conversion method based on finding the least positive solution (LPS) to multiple RNS systems of congruence equations. The conversion method presented herein produces a hardware implementation that is orders of magnitude smaller in terms of transistor count than the ROM and decoder conversion methods, and is easily pipelined to achieve fast conversion speeds. The following sections reveal the congruence equations for resolving the unknown incoming value from the RSNS residues, identify the crucial RSNS to RNS relationship, and finally describe the RSNS-to-binary LPS solution method.

1. RSNS Congruence Equations

Before presenting the congruence equations for converting the RSNS residues to binary, it is necessary to review the structure of the three-modulus RSNS from Chapter IV shown in Figure 5.4.

X_h	$(m_1 = 3)$	0	0	0	1	1	1	2	2	2	3	3	3	2	2	2	1	1	1	0	0	0	1	1	1	…
	$(m_2 = 4)$	0	0	1	1	1	2	2	2	3	3	3	4	4	4	3	3	3	2	2	2	1	1	1	0	…
	$(m_3 = 5)$	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	4	4	4	3	3	3	2	2	…
	h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…

Figure 5.4 Three-modulus RSNS structure with $\begin{bmatrix} m_1 & m_2 & m_3 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}^{\mathrm{T}}$.

Figure 5.5 illustrates decimating a single MRS $(m_1 = 3)$ into three MRSSs. Each MRSS is composed of values from the original MRS at positions where $h \equiv 0 \pmod 3$, $h \equiv 1 \pmod 3$, and $h \equiv 2 \pmod 3$.

$m_1 = 3$	x_h	0	0	0	1	1	1	2	2	2	3	3	3	2	2	2	1	1	1	0	0	0	1	…
$h = 0 \pmod 3$	x_h	0			1			2			3			2			1			0			1	…
$h = 1 \pmod 3$	x_h		0			1			2			3			2			1			0			…
$h = 2 \pmod 3$	x_h			0			1			2			3			2			1			0		…
	h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	…

Figure 5.5 Single MRS RSNS structure showing three MRSSs.

The top row in the figure is the original MRS and below it are the three MRSSs and the position index *h*. Notice that the positions of the MRSS integers do not align with each other. Every integer value *h* falls on one and only one MRSS. The position index *h* of each MRSS can be re-indexed to simplify the congruence equations, where the relationship between the old index *h* and the new index g is given by

$$g = \frac{h}{3}, \tag{5.2}$$

for the first MRSS,

$$g = \frac{(h-1)}{3}, \tag{5.3}$$

for the second MRSS, and

$$g = \frac{(h-2)}{3}, \tag{5.4}$$

for the third MRSS. The RSNS vectors for the three re-indexed MRSSs are shown in Figure 5.6, Figure 5.7, and Figure 5.8.

$m_1 = 3$		0	1	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2	…
$m_2 = 4$	X_h	0	1	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	…
$m_3 = 5$		0	1	2	3	4	5	4	3	2	1	0	1	2	3	4	5	4	…
h		0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	…
$g = h/3$		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…

Figure 5.6 Three-modulus RSNS vectors in the 0^{th} MRSS.

$m_1 = 3$		0	1	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2 …
$m_2 = 4$	X_h	0	1	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0 …
$m_3 = 5$		1	2	3	4	5	4	3	2	1	0	1	2	3	4	5	4	3 …
h		1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46	49 …
$g = (h-1)/3$		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 …

Figure 5.7 Three-modulus RSNS vectors in the 1st MRSS.

$m_1 = 3$		0	1	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2 …
$m_2 = 4$	X_h	1	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	1 …
$m_3 = 5$		1	2	3	4	5	4	3	2	1	0	1	2	3	4	5	4	3 …
h		2	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47	50 …
$g = (h-2)/3$		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 …

Figure 5.8 Three-modulus RSNS vectors in the 2nd MRSS.

A three-modulus RSNS vector is denoted

$$X_h = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}, \tag{5.5}$$

where the RSNS residues are in the range

$$\begin{aligned} s_1 &= \{0, 1, \cdots, m_1\}, \\ s_2 &= \{0, 1, \cdots, m_2\}, \\ s_3 &= \{0, 1, \cdots, m_3\}. \end{aligned} \tag{5.6}$$

Because of the structure of the RSNS MRSS shown in the previous figures, each residue integer except 0 and m_i occurs twice in a single period. Using this fact, the congruence equations describing the position of a residue vector for the 0th MRSS are

$$\begin{aligned} g &\equiv s_1 \pmod{2m_1} \quad \text{or} \quad g \equiv 2m_1 - s_1 \pmod{2m_1}, \\ g &\equiv s_2 \pmod{2m_2} \quad \text{or} \quad g \equiv 2m_2 - s_2 \pmod{2m_2}, \\ g &\equiv s_3 \pmod{2m_3} \quad \text{or} \quad g \equiv 2m_3 - s_3 \pmod{2m_3}. \end{aligned} \tag{5.7}$$

The congruence equations for the 1st MRSS are

$$\begin{aligned} g &\equiv s_1 \pmod{2m_1} \quad \text{or} \quad g \equiv 2m_1 - s_1 \pmod{2m_1}, \\ g &\equiv s_2 \pmod{2m_2} \quad \text{or} \quad g \equiv 2m_2 - s_2 \pmod{2m_2}, \\ g &\equiv s_3 - 1 \pmod{2m_3} \quad \text{or} \quad g \equiv 2m_3 - s_3 - 1 \pmod{2m_3}. \end{aligned} \tag{5.8}$$

The congruence equations for the 2nd MRSS are

$$\begin{aligned} g &\equiv s_1 \pmod{2m_1} \quad \text{or} \quad g \equiv 2m_1 - s_1 \pmod{2m_1}, \\ g &\equiv s_2 - 1 \pmod{2m_2} \quad \text{or} \quad g \equiv 2m_2 - s_2 - 1 \pmod{2m_2}, \\ g &\equiv s_3 - 1 \pmod{2m_3} \quad \text{or} \quad g \equiv 2m_3 - s_3 - 1 \pmod{2m_3}. \end{aligned} \tag{5.9}$$

The goal, given a RSNS residue vector $X_h = \begin{bmatrix} s_1 & s_2 & s_3 \end{bmatrix}^{\mathrm{T}}$, is to find the index g by solving the systems of congruence equations in (5.7), (5.8), and (5.9). The index g can then be converted to the incoming value *h* using (5.2), (5.3), or (5.4) based on the MRSS of the residue vector X_h. Because there are three rows of congruence equations for each MRSS (5.7), (5.8), and (5.9), with two possible choices in each row, each residue vector can produce up to eight systems of congruence equations. This means that any particular vector of residues $X_h = \begin{bmatrix} s_1 & s_2 & s_3 \end{bmatrix}^{\mathrm{T}}$ can have up to eight redundancies within the fundamental period of a three-modulus RSNS. Recall that the fundamental period for the three-modulus is [30]

$$P_f = 6m_1m_2m_3 . \tag{5.10}$$

Note, however, that if a particular MRS residue is a maximum $\left(s_i = m_i\right)$ or a minimum $\left(s_i = 0\right)$ then the congruence equation choices corresponding to the MRS with the maximum or minimum reduce to a single equation

$$g \equiv s_i \pmod{2m_i}, \tag{5.11}$$

or

$$g \equiv s_i - 1 \pmod{2m_i}, \tag{5.12}$$

depending on the form of the original MRS equation in (5.7), (5.8), or (5.9). Therefore, the number of vector redundancies in a fundamental period is reduced by a factor of two for each MRS that contains a maximum or minimum.

In order to determine which set of congruence equations (5.7), (5.8), or (5.9) to apply to a particular residue vector, it is necessary to scrutinize the residue vectors in Figure 5.6. A noticeable even-odd pattern occurs in the residue vectors. The residue vectors at even indexes are all even and the residue vectors at odd indexes are all odd, as shown in Figure 5.9. Therefore, the 0th MRSS equations (5.7) are applied to the residue vector when the residues are either all even or all odd. Extending this analysis to the other two MRSSs, the 1st MRSS equations (5.8) are applied when the X_h vector residue pattern is $[e\ e\ o]^{\mathrm{T}}$ or $[o\ o\ e]^{\mathrm{T}}$, and the 2nd MRSS equations (5.9) are applied when the X_h vector residue pattern is $[e\ o\ o]^{\mathrm{T}}$ or $[o\ e\ e]^{\mathrm{T}}$.

$m_1 = 3$		e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	…
$m_2 = 4$	X_h	e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	…
$m_3 = 5$		e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	o	e	…
h		0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	…
$g = h/3$		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…

Figure 5.9 Even/odd residue pattern for the 0th MRSS vectors.

Note that residue vectors with the pattern $[e\ o\ e]^{\mathrm{T}}$ or $[o\ e\ o]^{\mathrm{T}}$ do not exist in the MRSSs shown in this chapter. This is because the 1st, 2nd, and 3rd MRSs are shifted left by zero, one, and two positions, respectively, while each residue is repeated three times in each MRS. Since the MRS shifts are consecutive and their sum equals the number of residue repetitions, there will always be at least two adjacent residues with the same parity in each three-modulus RSNS vector. If the RSNS was formed by shifting the MRSs in a different manner, the vector patterns $[e\ o\ e]^{\mathrm{T}}$ or $[o\ e\ o]^{\mathrm{T}}$ could exist and would correspond to a particular MRSS.

The previous concepts are illustrated by the following three examples. Given an RSNS with the moduli in (5.1), the residue vector $X_h = \begin{bmatrix} 1 & 3 & 3 \end{bmatrix}^{\mathrm{T}}$ is from the 0^{th} MRSS because it is $\begin{bmatrix} o & o & o \end{bmatrix}^{\mathrm{T}}$ and contains no maximums or minimums. Applying this residue vector to the congruence equations from (5.7) yields

$$\begin{aligned} g &\equiv 1 \pmod 6 \quad \text{or} \quad g \equiv 5 \pmod 6, \\ g &\equiv 3 \pmod 8 \quad \text{or} \quad g \equiv 5 \pmod 8, \\ g &\equiv 3 \pmod{10} \quad \text{or} \quad g \equiv 7 \pmod{10}. \end{aligned} \tag{5.13}$$

Choosing one congruence equation from each row of (5.13) forms eight unique systems of congruence equations:

$$\begin{aligned} g &\equiv 1 \pmod 6, \\ g &\equiv 3 \pmod 8, \\ g &\equiv 3 \pmod{10}, \end{aligned} \tag{5.14}$$

$$\begin{aligned} g &\equiv 1 \pmod 6, \\ g &\equiv 3 \pmod 8, \\ g &\equiv 7 \pmod{10}, \end{aligned} \tag{5.15}$$

$$\begin{aligned} g &\equiv 1 \pmod 6, \\ g &\equiv 5 \pmod 8, \\ g &\equiv 3 \pmod{10}, \end{aligned} \tag{5.16}$$

$$\begin{aligned} g &\equiv 1 \pmod 6, \\ g &\equiv 5 \pmod 8, \\ g &\equiv 7 \pmod{10}, \end{aligned} \tag{5.17}$$

$$\begin{aligned} g &\equiv 5 \pmod 6, \\ g &\equiv 3 \pmod 8, \\ g &\equiv 3 \pmod{10}, \end{aligned} \tag{5.18}$$

$$\begin{aligned} g &\equiv 5 \pmod 6, \\ g &\equiv 3 \pmod 8, \\ g &\equiv 7 \pmod{10}, \end{aligned} \tag{5.19}$$

$$\begin{aligned} g &\equiv 5 \pmod 6, \\ g &\equiv 5 \pmod 8, \\ g &\equiv 3 \pmod{10}, \end{aligned} \tag{5.20}$$

$$\begin{aligned} g &\equiv 5 \pmod 6, \\ g &\equiv 5 \pmod 8, \\ g &\equiv 7 \pmod{10}. \end{aligned} \tag{5.21}$$

Each system of congruence equations can be solved using the general form of the CRT, producing a total of eight unique solutions. After converting the solution g to the index h using (5.2), each solution is a position of the vector $X_h = \begin{bmatrix} 1 & 3 & 3 \end{bmatrix}^{\mathrm{T}}$ within the RSNS fundamental period.

As a second example, the residue vector $X_h = \begin{bmatrix} 0 & 4 & 4 \end{bmatrix}^{\mathrm{T}}$ is also from the 0^{th} MRSS because it is $\begin{bmatrix} e\, e\, e \end{bmatrix}^{\mathrm{T}}$. It contains a minimum residue $(s_1 = 0)$ and a maximum residue $(s_2 = 4)$. Therefore, the congruence equations for this residue vector using (5.7) are

$$\begin{aligned} g &\equiv 0 \pmod 6, \\ g &\equiv 4 \pmod 8, \\ g \equiv 4 \pmod{10} \quad &\text{or} \quad g \equiv 6 \pmod{10}, \end{aligned} \tag{5.22}$$

which produce only two systems of three congruence equations and consequently have only two unique solutions within the RSNS fundamental period.

As a final example, the residue vector $X_h = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$ is also in the 0^{th} MRSS and the residues are minimums. The congruence equations are

$$\begin{aligned} g &\equiv 0 \pmod 6, \\ g &\equiv 0 \pmod 8, \\ g &\equiv 0 \pmod{10}, \end{aligned} \tag{5.23}$$

and there is only one solution within the fundamental period, which happens to be at $g = h = 0$.

The discussion thus far has produced a procedure to find the unknown incoming value *h* that corresponds to the residues $X_h = [s_1\ s_2\ s_3]^{\mathrm{T}}$ within $\hat{M}$. The procedure is:

- Select the set of congruence equation choices from (5.7), (5.8), or (5.9) based on the even-odd pattern of the RSNS residue vector.
- Expand the congruence equation choices into one, two, four, or eight systems of congruence equations.
- Compute the solution to each system of congruence equations using the generalized form of the CRT to find the index *g*.
- Convert index *g* to index *h* using (5.2), (5.3), or (5.4) based on the even-odd pattern of the RSNS vector residues.
- The unique solution *h* that is within $\hat{M}$ is the only position of the residue vector $X_h = [s_1\ s_2\ s_3]^{\mathrm{T}}$ within $\hat{M}$.

As an example, consider the three-modulus RSNS with the moduli in (5.1) where the lower bound of $\hat{M}$ is $h = 79$ and the upper bound is $h = 121$ [30]. Given the residue vector $X_h = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}^{\mathrm{T}}$, the goal is to find the *h* value within $\hat{M}$ that corresponds to this particular residue vector. Following the procedure above, the first step is to recognize that the residue vector has the pattern $\begin{bmatrix} e & e & o \end{bmatrix}^{\mathrm{T}}$, which means it is from the 1st MRSS. Thus, applying the 1st MRSS congruence equations from (5.8) yields

$$\begin{aligned} &g \equiv 2 \quad or \quad g \equiv 4 \pmod 6, \\ &g \equiv 2 \quad or \quad g \equiv 6 \pmod 8, \\ &g \equiv 2 \quad or \quad g \equiv 6 \pmod{10}. \end{aligned} \tag{5.24}$$

The congruence equation choices in (5.24) can be expanded to form eight unique systems of congruence equations. The solution to each system of congruence equations can be found using the CRT, yielding the index *g*. Applying the 1st MRSS index conversion $h = 3g + 1$ from (5.3) to the solutions for index *g* yields all positions of the RSNS residue vector $X_h = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}^{\mathrm{T}}$ in the fundamental period and are shown in Figure 5.10. Since the vector $X_h = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}^{\mathrm{T}}$ has a redundancy at $h = 79$, which is within $\hat{M}$, $h = 79$ is the unknown position of the residue vector within $\hat{M}$ (mod $P_f = 360$).

$$
\begin{array}{l}
g \equiv 2 \pmod 6 \\
g \equiv 2 \pmod 8 \quad \rightarrow \quad h = 7 \\
g \equiv 2 \pmod{10} \\
\\
g \equiv 2 \pmod 6 \\
g \equiv 2 \pmod 8 \quad \rightarrow \quad h = 79 \\
g \equiv 6 \pmod{10} \\
\\
g \equiv 2 \pmod 6 \\
g \equiv 6 \pmod 8 \quad \rightarrow \quad h = 187 \\
g \equiv 2 \pmod{10} \\
\\
g \equiv 2 \pmod 6 \\
g \equiv 6 \pmod 8 \quad \rightarrow \quad h = 259 \\
g \equiv 6 \pmod{10} \\
\\
g \equiv 4 \pmod 6 \\
g \equiv 2 \pmod 8 \quad \rightarrow \quad h = 247 \\
g \equiv 2 \pmod{10} \\
\\
g \equiv 4 \pmod 6 \\
g \equiv 2 \pmod 8 \quad \rightarrow \quad h = 319 \\
g \equiv 6 \pmod{10} \\
\\
g \equiv 4 \pmod 6 \\
g \equiv 6 \pmod 8 \quad \rightarrow \quad h = 67 \\
g \equiv 2 \pmod{10} \\
\\
g \equiv 4 \pmod 6 \\
g \equiv 6 \pmod 8 \quad \rightarrow \quad h = 139 \\
g \equiv 6 \pmod{10}
\end{array}
$$

Figure 5.10 Eight redundancy solutions for the residue vector $X_h = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}^{\mathrm{T}}$.

Unfortunately, it is extremely hardware intensive to simultaneously solve eight generalized CRT solutions for index *g*, convert to index *h*, and compare the results to the $\hat{M}$ lower and upper bounds. Therefore, the formulation of an alternative approach is necessary and requires a closer examination of the solutions to the congruence equations (5.7), (5.8), and (5.9).

The congruence equations (5.8) and (5.9) for the X_h residue vectors of the form $[e\ e\ o]^{\mathrm{T}}$, $[o\ o\ e]^{\mathrm{T}}$, $[o\ e\ e]^{\mathrm{T}}$, and $[e\ o\ o]^{\mathrm{T}}$ actually convert the X_h residue vector to the form $[e\ e\ e]^{\mathrm{T}}$ or $[o\ o\ o]^{\mathrm{T}}$. Equation (5.24) in the example above provides an illustration of this concept. Moreover, the 1st and 2nd MRSS residue vectors are converted to the *closest* 0th MRSS $[e\ e\ e]^{\mathrm{T}}$ or $[o\ o\ o]^{\mathrm{T}}$ vector with the smallest *h* index. This means that the solution for the 1st MRSS and 2nd MRSS residue vectors can be found by using (5.7) to compute the *h* index of the converted 0th MRSS residue vector, then incrementing the 0th MRSS solution by one or two depending on the MRSS of the original X_h residue vector. As a result, the following sections focus solely on developing an efficient solution to the 0th MRSS equations (5.7).

2. The RSNS-RNS Relationship

To facilitate the development of an efficient solution to (5.7), it is necessary to review a few terms and define some new ones. The vector $X_h = [s_1\ s_2\ s_3]^{\mathrm{T}}$ is the RSNS residue vector at index *h*. Each residue vector in the 0th MRSS has the form $[e\ e\ e]^{\mathrm{T}}$ or $[o\ o\ o]^{\mathrm{T}}$. Each residue vector in the 0th MRSS, when combined with (5.7), forms up to eight systems of congruence equations. Writing (5.7) in vector form results in

$$g \equiv \begin{bmatrix} s_1 & \text{or} & 2m_1 - s_1 \\ s_2 & \text{or} & 2m_2 - s_2 \\ s_3 & \text{or} & 2m_3 - s_3 \end{bmatrix} \left(\text{mod } 2 \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} \right), \tag{5.25}$$

or simply

$$g \equiv XR_g \left(\text{mod } 2 \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} \right), \tag{5.26}$$

where XR_g is the residue number system vector representing a *single* system of congruence equations, meaning each $(s_i \text{ or } 2m_i - s_i)$ choice has been made. The solution to the system of congruence equations (5.26) is the index g, and is one of the redundant positions of the residue vector $X_g = [s_1 \; s_2 \; s_3]^T$ in the 0th MRSS. Note that the vector $X_g = [s_1 \; s_2 \; s_3]^T$ in the 0th MRSS is the same vector as $X_h = [s_1 \; s_2 \; s_3]^T$ in the RSNS, where g is related to h by (5.2). Therefore, the key concept from this analysis is that every vector X_g in the 0th MRSS can be combined with (5.25) to form a system of congruence equations uniquely represented by (5.26) and the vector XR_g that has a solution of g. Figure 5.11 shows the vectors X_g and the corresponding vectors XR_g at each index g.

$m_1 = 3$		0	1	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2	…
$m_2 = 4$	X_g	0	1	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	…
$m_3 = 5$		0	1	2	3	4	5	4	3	2	1	0	1	2	3	4	5	4	…
		0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	…
	XR_g	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	…
		0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	…
	g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…

Figure 5.11 Vectors X_g and XR_g for the 0th MRSS.

Notice that the rows in the figure corresponding to XR_g vectors form a *residue number system* (as opposed to a *symmetrical* residue number system) with moduli $2m_i$, where the index i is the MRS number. This is a remarkable discovery since it demonstrates a direct relationship between the RSNS and the RNS. The RNS has no ambiguities in a fundamental period and is well studied while the RSNS is relatively new and has somewhat unpredictable ambiguities. Another useful observation is that each residue vector XR_g is either all odd or all even and has the relationship

$$XR_{g+1} \equiv \left(XR_g + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) \left(\text{mod } 2 \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} \right). \tag{5.27}$$

Therefore, it is possible to exploit this structure by decimating the RNS vectors by a factor of two. Dividing the vector XR_g, index g, and the system moduli by two creates the RNS vectors shown in Figure 5.12.

RSNS Vectors	$X_{g/2}$	0	2	2	0	2	2	0	2	2	…
		0	2	4	2	0	2	4	2	0	…
		0	2	4	4	2	0	2	4	4	…
RNS Vectors (PRP moduli)	$\frac{XR_{g/2}}{2}$	0	1	2	0	1	2	0	1	2	…
		0	1	2	3	0	1	2	3	0	…
		0	1	2	3	4	0	1	2	3	…
	$g/2$	0	1	2	3	4	5	6	7	8	…

Figure 5.12 Vectors $X_{g/2}$ and $XR_{g/2}/2$ with index $g/2$.

Consequently, the RSNS residue vectors X_h are transformed into RNS residue vectors with PRP moduli, whose index positions can be solved using the CRT. Alas, there are still up to eight systems of RNS congruence equations to solve for each RSNS residue vector. Fortunately, using the lower and upper bounds of $\hat{M}$ found by means of the search algorithm developed in Chapter IV, the problem of finding the solution within $\hat{M}$ is reduced to finding the least positive solution of the eight systems of congruence equations. This concept will be explained in the next section.

3. RNS Least Positive Solution

The longest sequence of unique RSNS vectors for moduli given in (5.1) occurs from $h = 79$ to $h = 121$, inclusive [30]. These boundaries correspond to RSNS residue vectors $X_{79} = [2 \quad 2 \quad 3]^T$ and $X_{121} = [2 \quad 0 \quad 1]^T$. The using the procedure described in

the previous section, the closest 0th MRSS vectors are $X_{78} = [2 \quad 2 \quad 4]^T$ and $X_{120} = [2 \quad 0 \quad 0]^T$. Using (5.2), the index *h* is converted to index *g* and the vectors X_g within $\hat{M}$ for the 0th MRSS are displayed as the first three rows of Figure 5.13. The systems of congruence equations formed using the two boundary vectors X_g are represented by the RNS vectors $XR_{26} = [2 \quad 2 \quad 6]^T$ and $XR_{40} = [4 \quad 0 \quad 0]^T$ and are displayed in the next three rows of Figure 5.13.

$m_1 = 3$		…	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2	…
$m_2 = 4$	X_g	…	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	…
$m_3 = 5$		…	4	3	2	1	0	1	2	3	4	5	4	3	2	1	0	…
		…	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	…
	XR_g	…	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	…
		…	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	…
	g	…	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	…

Figure 5.13 The 0th MRSS residue vectors within $\hat{M}$.

Subtracting the vector $XR_{26} = [2 \quad 2 \quad 6]^T$ from each XR_g vector in Figure 5.13 (mod M_{345}) effectively shifts the lower bound of $\hat{M}$ from an arbitrary position in the fundamental period down to $g = 0$ as shown in Figure 5.14.

$m_1 = 3$		…	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2	…
$m_2 = 4$	X_g	…	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	…
$m_3 = 5$		…	4	3	2	1	0	1	2	3	4	5	4	3	2	1	0	…
		…	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	…
	XR_g	…	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	…
		…	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	…
	g	…	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Figure 5.14 The 0th MRSS residue vectors within $\hat{M}$ shifted to $g = 0$.

This makes the location of the residue vector $X_g = [s_1\ s_2\ s_3]^{\mathrm{T}}$ that falls within $\hat{M}$ equal to the least positive solution of the eight systems of congruence equations. Furthermore, the 0^{th} MRSS solution position g, and consequently the RSNS position h is found relative to the lower bound of $\hat{M}$ rather than a seemingly arbitrary position in the RSNS fundamental period. Thus, the solution will be in the range $1 \le h \le 43$ rather than $79 \le h \le 121$, which is desirable since the former can be contained in a 6-bit binary number and the latter requires 7 bits.

A method of finding the least positive solution to multiple systems of congruence equations that is easily adaptable to a hardware implementation is a *positional alignment solution technique*. A positional alignment solution technique symbolically marks the positions of the RNS residues on the XR_g vectors in Figure 5.14.

The following example illustrates the positional alignment solution technique. Given the RSNS residue vector $X_h = \begin{bmatrix} 0 & 2 & 1 \end{bmatrix}^{\mathrm{T}}$, and an $\hat{M}$ with bounds $79 \le h \le 121$, the goal is to find the position of X_h within the bounds of $\hat{M}$. Since X_h is $\begin{bmatrix} e & e & o \end{bmatrix}^{\mathrm{T}}$, it is a 1^{st} MRSS vector and (5.8) is used to convert the X_h RSNS residues to XR_g RNS residues that form the congruence equations

$$\begin{gathered} g \equiv 0 \pmod 6, \\ g \equiv 2 \pmod 8 \quad \text{or} \quad g \equiv 6 \pmod 8, \\ g \equiv 0 \pmod{10} \quad \text{or} \quad g \equiv 8 \pmod{10}. \end{gathered} \tag{5.28}$$

The RNS residue choices from (5.28) for each MRS from (5.24) are highlighted in gray in Figure 5.15.

$m_1 = 3$		…	2	3	2	1	0	1	2	3	2	1	0	1	2	3	2	…
$m_2 = 4$	X_g	…	2	3	4	3	2	1	0	1	2	3	4	3	2	1	0	…
$m_3 = 5$		…	4	3	2	1	0	1	2	3	4	5	4	3	2	1	0	…
		…	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	…
	XR_g	…	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	…
		…	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	…
	g	…	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	…

Figure 5.15 RNS residues highlighted on 0th MRSS $\hat{M}$ vectors.

The position at which the gray residue blocks align is at $g = 30$, which is the correct solution within $\hat{M}$. Applying (5.2) converts $g = 30$ to $h = 90$. This solution is a 0th MRSS solution, and since $X_h = \begin{bmatrix} 0 & 2 & 1 \end{bmatrix}^{\mathrm{T}}$ is $\begin{bmatrix} e & e & o \end{bmatrix}^{\mathrm{T}}$ it is a 1st MRSS vector. The 0th MRSS solution must be incremented by one to compute the 1st MRSS solution. Therefore, $h = 91$ is the final and correct solution. Ambiguous results can occur using this method if $\hat{M}$ does not start on a 0th MRSS boundary and end on a 2nd MRSS boundary. Therefore, $\hat{M}$ should either be truncated accordingly or the boundary vectors must be treated as special cases in a hardware implementation. Examples of both scenarios are provided in later sections.

The shaded blocks in Figure 5.15 further illustrate the concept introduced in Figure 5.12. Notice in the example that $s_1 = 0$ (i.e., s_1 has even parity) and the shaded blocks occur only in the even g positions. If s_1 was odd instead of even (e.g., in the vector $X_h = \begin{bmatrix} 1 & 1 & 3 \end{bmatrix}^{\mathrm{T}}$), then all the shaded residues would occur in the odd g positions. Thus, if s_1 is odd, the odd XR_g vector can be converted to the even XR_g vector by subtracting $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$ from the odd XR_g vector following (5.27). The information regarding the parity of s_1 is retained to reverse the compression later in the conversion process. This reduces the number of unique solution vectors by a factor of two like in Figure 5.12, leaving only the RNS solution vectors in Figure 5.16.

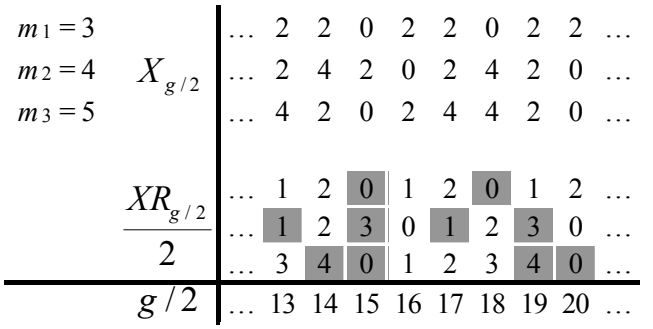

Figure 5.16 Residues highlighted on $XR_{g/2}/2$ vectors index $g/2$.within $\hat{M}$ in the 0^{th} MRSS.

The total $\hat{M}$ compression factor in this example is six. That is, using only the 0^{th} MRSS compresses the number of vectors in $\hat{M}$ by three, and exploiting the parity of s_1 yields an additional compression factor of two. Thus, given the residue vector MRSS number and the parity of s_1, only

$$\left\lceil \frac{\hat{M}}{6} \right\rceil = \left\lceil \frac{43}{6} \right\rceil = 8, \tag{5.29}$$

RNS vectors are needed to find the LPS solution for the index *g*. In the general case, the compression factor is $2N$ because there are *N* MRSSs in the general case, with the additional factor of 2 stemming from the knowledge of the parity of s_1. Thus, the number of RNS vectors required to find *g* in the general case is

$$\left\lceil \frac{\hat{M}}{2N} \right\rceil. \tag{5.30}$$

For a comprehensive example combining all the previous concepts, consider the residue vector $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^{\mathrm{T}}$, which has the form $\begin{bmatrix} o & e & e \end{bmatrix}^{\mathrm{T}}$ and is a 2^{nd} MRSS vector. Using (5.9), the congruence equations are

$$g \equiv \begin{bmatrix} 1 \text{ or } 5 \\ 3 \\ 3 \text{ or } 5 \end{bmatrix} \left(\text{mod} \begin{bmatrix} 6 \\ 8 \\ 10 \end{bmatrix} \right). \tag{5.31}$$

Equation (5.31) expands to only four systems of congruence equations since s_2 in X_h is a maximum. Given that all of the residues in (5.31) are odd, they are all converted to even residues by subtracting $\begin{bmatrix}1 & 1 & 1\end{bmatrix}^{\mathrm{T}}$ (mod M_{345}) and the odd parity of s_1 is recorded for later use. The resulting congruence equations are

$$g \equiv \begin{bmatrix} 0 \text{ or } 4 \\ 2 \\ 2 \text{ or } 4 \end{bmatrix} \left(\text{mod} \begin{bmatrix} 6 \\ 8 \\ 10 \end{bmatrix} \right). \tag{5.32}$$

Dividing both sides of (5.32) by two represents the factor of two compression since the residue vectors are reduced to the nearest even 0^{th} MRSS vectors with positions less than or equal to the original index *h*. The resulting congruence equations are

$$\frac{g}{2} \equiv \begin{bmatrix} 0 \text{ or } 2 \\ 1 \\ 1 \text{ or } 2 \end{bmatrix} \left(\text{mod} \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \right). \tag{5.33}$$

From Figure 5.16, the lower bound of $\hat{M}$ is $XR_{26}/2 = [1 \quad 1 \quad 3]^{\mathrm{T}}$. Subtracting this vector from the residues in (5.33) (mod M_{345}) shifts the lower bound of $\hat{M}$ down to an index of $g/2 = 0$ and yields

$$\frac{g}{2} \equiv \begin{bmatrix} 2 \text{ or } 1 \\ 0 \\ 3 \text{ or } 4 \end{bmatrix} \left(\text{mod} \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \right). \tag{5.34}$$

Shading the residues from (5.34) on the RNS solution vectors within $\hat{M}$ from Figure 5.12 results in Figure 5.17.

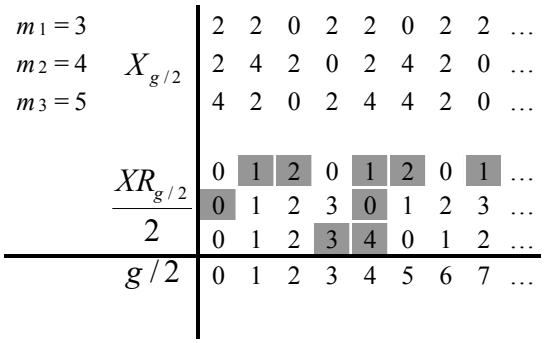

Figure 5.17 Positional solution for (5.34).

The solution that falls within $\hat{M}$ is found at position $g/2 = 4$ where all the gray shaded blocks align. To compute the unknown position *h* corresponding to the RSNS example vector $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^{\mathrm{T}}$, the even-odd compression and MRSS compression must be reversed. Defining F_{odd} as the even-odd compensation factor, and F_{MRSS} as the MRSS compensation factor, and using relationship between *g* and *h* given by (5.2), the expression for *h* is

$$h = 3(g + F_{odd}) + F_{MRSS}, \tag{5.35}$$

where

$$F_{odd} = \begin{Bmatrix} 0 & \text{if } s_1 \text{ is even} \\ 1 & \text{if } s_1 \text{ is odd} \end{Bmatrix},$$

$$F_{MRSS} = \begin{Bmatrix} 0 & \text{if } 0^{\text{th}} \text{ MRSS} \\ 1 & \text{if } 1^{\text{st}} \text{ MRSS} \\ 2 & \text{if } 2^{\text{nd}} \text{ MRSS} \end{Bmatrix}. \tag{5.36}$$

For $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^{\mathrm{T}}$, $F_{odd} = 1$ because s_1 is odd, and $F_{MRSS} = 2$ because X_h is from the 2^{nd} MRSS. Therefore, the solution to (5.35) is

$$h = 3(8 + 1) + 2 = 29, \tag{5.37}$$

which is the correct solution for the residue vector $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^{\mathrm{T}}$ within $\hat{M}$, where $1 \le h \le 43$. Figure 5.18 summarizes the LPS RSNS-to-binary conversion method.

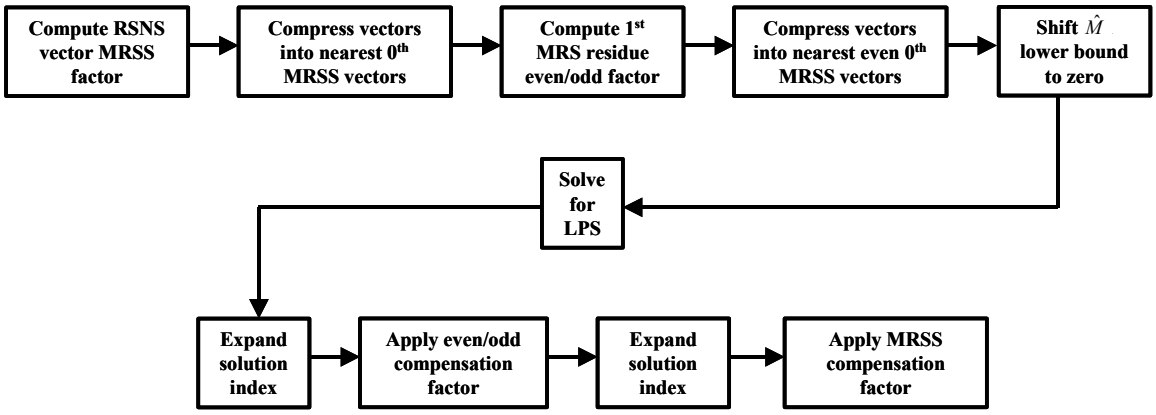

Figure 5.18 Summary of LPS RSNS-to-binary conversion method.

C. THREE-MODULUS RSNS-TO-BINARY LPS CONVERTER

The hardware implementation of the RSNS-to-binary LPS method developed in the previous section is realized in this section. The process is essentially the same as Figure 5.18. The first step is the RSNS to RNS residue compression and the F_{odd} and F_{MRSS} computation. In this step, the RNS residues and compensation factors are converted from integers to a digital representation. The second step is the implementing the $\hat{M}$ shift and computing the least positive positionally aligned solution to the eight systems of congruence equations. The third step is the conversion from the positional least positive solution to the unknown incoming value. The output is in the range $1 \le h \le 43$ and therefore can be represented by a 6-bit binary value.

1. RSNS Thermometer Code to RNS Residue Conversion

The first step in the conversion process is to translate the RSNS residues X_h to RNS residues XR_g and compute the even-odd and MRSS compensation factors F_{odd} and F_{MRSS}. Since the moduli are small in this case, it is feasible to convert the RSNS thermometer code residues to RNS residues using logic-minimizing Karnaugh maps. The notation for this section is as follows. The vector $X_h = [s_1\ s_2\ s_3]^T$ is the RSNS residue

vector and corresponds to the position *h* within $\hat{M}$. Individual residues themselves are encoded in a thermometer code. Consequently, the residue s_i, which is in the range of $0 \le s_i \le m_i$, is represented in thermometer code with m_i bits. Figure 5.19 shows the thermometer code bits for s_1 $(m_1 = 3)$.

RSNS residue value	RSNS thermometer code bits		
s_1	s_{12}	s_{11}	s_{10}
0	0	0	0
1	0	0	1
2	0	1	1
3	1	1	1

Figure 5.19 RSNS thermometer code bits for s_1 $(m_1 = 3)$.

Each of the bits in the thermometer code is labeled s_{ib}, where the index *i* represents the MRS in the RSNS residue vector, and the index *b* is the bit position in the thermometer code, with $b = 0$ corresponding to the least significant bit position. Figure 5.20 shows the thermometer code for the s_2 and s_3 ($m_2 = 4$ and $m_3 = 5$).

RSNS residue value	RSNS thermometer code bits				RSNS residue value	RSNS thermometer code bits				
s_2	s_{23}	s_{22}	s_{21}	s_{20}	s_3	s_{34}	s_{33}	s_{32}	s_{31}	s_{30}
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0	1
2	0	0	1	1	2	0	0	0	1	1
3	0	1	1	1	3	0	0	1	1	1
4	1	1	1	1	4	0	1	1	1	1
					5	1	1	1	1	1

Figure 5.20 RSNS thermometer code bits for s_2 and s_3 ($m_2 = 4$ and $m_3 = 5$).

The next step is the conversion of the RSNS residue to the corresponding RNS residue. The simplest case is the first MRS residue and will be considered first. Examination of (5.7), (5.8), and (5.9) reveals that the first line of each set of equations is identical. This means that the conversion of the first MRS RSNS residue to RNS residue is the same regardless of the MRSS. Figure 5.21 shows the conversion process for s_1 from RSNS to RNS.

RSNS Residue	**RNS Residue**		**Even RNS Residue**		**RNS Residue for PRP Moduli (r)**	
s_1	s_1	or $6-s_1$	even(s_1)	or even($6-s_1$)	$\frac{\text{even}(s_1)}{2}$	or $\frac{\text{even}(6-s_1)}{2}$
0	0	0	0	0	0	0
1	1	5	0	4	0	2
2	2	4	2	4	1	2
3	3	3	2	2	1	1

Figure 5.21 Conversion process from RSNS residue to RNS residue.

Starting with the first MRS RSNS residue s_1 shown in the first column of Figure 5.21, the RNS residues in the second and third column are found by application of the top equation from (5.7),

$$g \equiv s_1 \pmod{2m_1} \quad \text{or} \quad g \equiv 2m_1 - s_1 \pmod{2m_1}. \tag{5.38}$$

The odd RNS residues are then converted to the nearest even RNS residues by subtracting one as shown in columns four and five. Finally, the even RNS residues are divided by two, which form the RNS residues that correspond to PRP moduli shown in the last two columns of Figure 5.21. These RNS residues correspond to the top row of the $XR_{g/2}/2$ vector in Figure 5.17.

Since there are three possible RNS residue values in the last two columns of Figure 5.21, representing the RNS residues in digital format is accomplished using three *position bits*. Each unique RNS residue value *must* be represented by a *single* position bit because the hardware implementation of the positional alignment solution technique re-

quires physical alignment of the RNS residues. The position bits are denoted p_{ir}, where the index *i* is the MRS and *r* spans the range of the RNS residue values in the last two columns of Figure 5.21, which is $0 \le r \le (m_i - 1)$. The position bit corresponding to the RNS residue in either column is asserted. For example, the three position bits for the RSNS residue s_1 are p_{10}, p_{11}, and p_{12}. Looking at the second row of the last two columns of Figure 5.21, the two RNS residues are 0 and 2. Therefore, the position bits p_{10} and p_{12} would be asserted and p_{11} would be negated, thereby representing the RNS residues in digital form. Using this notation, Figure 5.22 shows the position bits corresponding to the RNS residues from Figure 5.21.

RSNS Residue	**RNS Residue for PRP Moduli (*r*)**		**Position Bits**		
s_1	$\frac{\text{even}(s_1)}{2}$	or $\frac{\text{even}(6 - s_1)}{2}$	p_{12}	p_{11}	p_{10}
0	0	0	0	0	1
1	0	2	1	0	1
2	1	2	1	1	0
3	1	1	0	1	0

Figure 5.22 Position bits corresponding to the RSNS residue s_1.

Next, the position bits for s_2 and s_3 are computed. The position bits for s_2 are only slightly more complicated to compute than the position bits for s_1. Notice that the form of the second row of the MRSS congruence equations in (5.7), (5.8), and (5.9) is not the same for all three sets of equations. This means that the position bits corresponding to s_2 will have one arrangement given a residue vector X_h from the 0th MRSS or 1st MRSS, and a different arrangement given a residue vector X_h from the 2nd MRSS. For a residue vector X_h, the 0th MRSS and 1st MRSS the RNS residues for s_2 are found by application of the second row of (5.7) and (5.8),

$$g \equiv s_2 \pmod{2m_2} \quad \text{or} \quad g \equiv 2m_2 - s_2 \pmod{2m_2}. \tag{5.39}$$

The conversion process from the s_2 residue to RNS residue (for a RSNS vector X_h from the 0th MRSS or 1st MRSS) is shown in Figure 5.23 and the corresponding position bits are shown in Figure 5.24.

RSNS Residue	**RNS Residue**		**Even RNS Residue**		**RNS Residue for PRP Moduli (r)**	
s_2	s_2	or $8-s_2$	$\text{even}(s_2)$	or $\text{even}(8-s_2)$	$\frac{\text{even}(s_2)}{2}$	or $\frac{\text{even}(8-s_2)}{2}$
0	0	0	0	0	0	0
1	1	7	0	6	0	3
2	2	6	2	6	1	3
3	3	5	2	4	1	2
4	4	4	4	4	2	2

Figure 5.23 Conversion process for s_2 from RSNS residue to RNS residue (for a RSNS vector X_h from the 0th MRSS or 1st MRSS).

RSNS Residue	**RNS Residue for PRP Moduli (r)**		**Position Bits for s_2 if X_h is in the 0th or 1st MRSS**			
s_2	$\frac{\text{even}(s_2)}{2}$	or $\frac{\text{even}(8-s_2)}{2}$	p_{23}	p_{22}	p_{21}	p_{20}
0	0	0	0	0	0	1
1	0	3	1	0	0	1
2	1	3	1	0	1	0
3	1	2	0	1	1	0
4	2	2	0	1	0	0

Figure 5.24 Position bits for s_2 (for a RSNS vector X_h from the 0th MRSS or 1st MRSS).

For the 2nd MRSS, the RSNS to RNS equation for s_2 that comes from the second row of (5.9) is

$$g \equiv s_2 - 1 \pmod{2m_2} \quad \text{or} \quad g \equiv 2m_2 - s_2 - 1 \pmod{2m_2}. \tag{5.40}$$

Using this equation, the conversion process for s_2 from RSNS residue to RNS residue (for a RSNS vector X_h from the 2nd MRSS) is shown in Figure 5.25 and the corresponding position bits are shown in Figure 5.26.

RSNS Residue	**RNS Residue**		**Even RNS Residue**		**RNS Residue for PRP Moduli (r)**	
s_2	$s_2 - 1$	or $7 - s_2$	$\text{even}(s_2 - 1)$	or $\text{even}(7 - s_2)$	$\frac{\text{even}(s_2 - 1)}{2}$	or $\frac{\text{even}(7 - s_2)}{2}$
0	7	7	6	6	3	3
1	0	6	0	6	0	3
2	1	5	0	4	0	2
3	2	4	2	4	1	2
4	3	3	2	2	1	1

Figure 5.25 Conversion process for s_2 from RSNS residue to RNS residue (for a RSNS vector X_h from the 2nd MRSS).

RSNS Residue	**RNS Residue for PRP Moduli (r)**		**Position Bits for s_2 if X_h is in the 2nd MRSS**			
s_2	$\frac{\text{even}(s_2 - 1)}{2}$	or $\frac{\text{even}(7 - s_2)}{2}$	p_{23}	p_{22}	p_{21}	p_{20}
0	3	3	1	0	0	0
1	0	3	1	0	0	1
2	0	2	0	1	0	1
3	1	2	0	1	1	0
4	1	1	0	0	1	0

Figure 5.26 Position bits for s_2 (for a RSNS vector X_h from the 2nd MRSS).

Comparing the position bits for s_2 in Figure 5.24 and Figure 5.26, it is apparent that the order of the s_2 position bits for the 2nd MRSS are the reverse of the 0th and 1st

MRSS position bits. This is illustrated in Figure 5.27. This fact will be very useful when constructing the circuit for the RSNS-to-binary converter.

RSNS Residue	**Position Bits for s_2 if X_h is in the 0^{th} or 1^{st} MRSS**				**Position Bits for s_2 if X_h is in the 2^{nd} MRSS**			
s_2	p_{23}	p_{22}	p_{21}	p_{20}	p_{23}	p_{22}	p_{21}	p_{20}
0	0	0	0	1	1	0	0	0
1	1	0	0	1	1	0	0	1
2	1	0	1	0	0	1	0	1
3	0	1	1	0	0	1	1	0
4	0	1	0	0	0	0	1	0

Figure 5.27 Position bits for s_2 (for a RSNS vector X_h from the 0^{th} MRSS, 1^{st} MRSS, or 2^{nd} MRSS).

The position bits for s_3 are found in exactly the same manner as s_1 and s_2. In this case, the RNS residue is found by application of the third row of (5.7), (5.8), and (5.9). Similar to the position bits for s_2, the position bit arrangement is different depending on the MRSS of the residue vector X_h. The position bits for s_3 (for a RSNS vector X_h from the 0^{th} MRSS, 1^{st} MRSS, or 2^{nd} MRSS) are shown in Figure 5.28. Similar to the position bits for s_2, the order of the position bits for s_3 for an RSNS vector X_h from the 1^{st} and 2^{nd} MRSSs are the reverse of the position bits for s_3 for an RSNS vector X_h from the 0^{th} MRSS.

RSNS Residue	Position Bits for s_3 if X_h is in the 0^{th} MRSS					Position Bits for s_3 if X_h is in the 1^{st} or 2^{nd} MRSS				
s_3	p_{34}	p_{33}	p_{32}	p_{31}	p_{30}	p_{34}	p_{33}	p_{32}	p_{31}	p_{30}
0	0	0	0	0	1	1	0	0	0	0
1	1	0	0	0	1	1	0	0	0	1
2	1	0	0	1	0	0	1	0	0	1
3	0	1	0	1	0	0	1	0	1	0
4	0	1	1	0	0	0	0	1	1	0
5	0	0	1	0	0	0	0	1	0	0

Figure 5.28 Position bits for s_3 (for a RSNS vector X_h from the 0^{th} MRSS, 1^{st} MRSS, or 2^{nd} MRSS).

Since the position bits uniquely correspond to the RNS residues for PRP moduli, they are used to compute the positional alignment solution in the hardware implementation of the RSNS-to-binary converter. In fact, the direct correspondence between the RNS residues and the position bits enable the position bits to replace the RNS residues in Figure 5.15 as shown in Figure 5.29. Notice that the first subscript in each position bit is the row of the RNS residue vector and the second subscript in each position bit is the same as the corresponding RNS residue.

$m_1 = 3$		2	2	0	2	2	0	2	2	…
$m_2 = 4$	$X_{g/2}$	2	4	2	0	2	4	2	0	…
$m_3 = 5$		4	2	0	2	4	4	2	0	…
	Position Bits	p_{11}	p_{12}	p_{10}	p_{11}	p_{12}	p_{10}	p_{11}	p_{12}	…
		p_{21}	p_{22}	p_{23}	p_{20}	p_{21}	p_{22}	p_{23}	p_{20}	…
		p_{33}	p_{34}	p_{30}	p_{31}	p_{32}	p_{33}	p_{34}	p_{30}	…
	$g/2$	0	1	2	3	4	5	6	7	…

Figure 5.29 Location of position bits in the compressed $\hat{M}$.

As an illustration of the use of the position bits to find the positionally aligned solution, consider the RSNS residue vector $X_h = [1 \quad 4 \quad 4]^T$ from the 2^{nd} MRSS and the corresponding congruence equations provided in (5.31)-(5.34). The values of the posi-

tion bits for s_1, s_2, and s_3 are extracted from Figure 5.22, Figure 5.24, and Figure 5.26 using the residue values in the vector $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^T$ and the 2nd MRSS columns in the case of s_2, and s_3. The positional solution is shown in Figure 5.30 with the asserted position bits highlighted in gray. This solution $(g/2 = 4)$ is the same as computed in Figure 5.17 using the same residue vector $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^T$.

$m_1 = 3$		2	2	0	2	2	0	2	2 …
$m_2 = 4$	$X_{g/2}$	2	4	2	0	2	4	2	0 …
$m_3 = 5$		4	2	0	2	4	4	2	0 …
		0	1	1	0	1	1	0	…
	Position		0	0	0	1	0	0	0 …
	Bits	0	0	0	1	1	0	0	0 …
	$g/2$	0	1	2	3	4	5	6	7 …

Figure 5.30 Least positive solution for $X_h = \begin{bmatrix} 1 & 4 & 4 \end{bmatrix}^T$ using position bits.

2. RNS Position Bit, Even Residue, and MRSS Logic Equations

Given the RSNS thermometer code bits and the corresponding position bits from Figure 5.22, Figure 5.24, and Figure 5.26, the minimized logic equations for computing the position bits can be found using logic tables and Karnaugh maps. Furthermore, error correction can be built in to the thermometer code to position bit conversion.

The only allowable combinations of thermometer code bits for the moduli given in (5.1) are shown in Figure 5.19 and Figure 5.20, which is less than all possible binary combinations for the number of thermometer bits. The combinations that are invalid thermometer codes can be treated as *don't cares* (X) in the logic table as shown in Figure 5.31.

1st MRS Residue					
RSNS Residue			Position Bits		
s_{12}	s_{11}	s_{10}	p_{12}	p_{11}	p_{10}
0	0	0	0	0	1
0	0	1	1	0	1
0	1	0	X	X	X
0	1	1	1	1	0
1	0	0	X	X	X
1	0	1	X	X	X
1	1	0	X	X	X
1	1	1	0	1	0

Figure 5.31 Logic table for computing the first MRS (s_1) position bits.

The unused combinations can also be treated as errors and corrected to the thermometer code that minimizes the *Hamming distance* [43] as shown in Figure 5.32.

1st MRS Residue										
RSNS Residue				Error Correction				Position Bits		
s_{12}	s_{11}	s_{10}		s_{12}	s_{11}	s_{10}		p_{12}	p_{11}	p_{10}
0	0	0						0	0	1
0	0	1						1	0	1
0	1	0	or	0	0	0		0	0	1
				0	1	1		1	1	0
0	1	1						1	1	0
1	0	0		0	0	0		0	0	1
1	0	1	or	0	0	1		1	0	1
				1	1	1		0	1	0
1	1	0		1	1	1		0	1	0
1	1	1						0	1	0

Figure 5.32 Logic table for computing the first MRS (s_1) position bits with error correction.

When the distance is the same between the residue error value and two different thermometer codes, the thermometer code that produces a simpler logic function is selected. Figure 5.33 and Figure 5.34 show the logic minimization for the logic tables in Figure 5.31 and Figure 5.32.

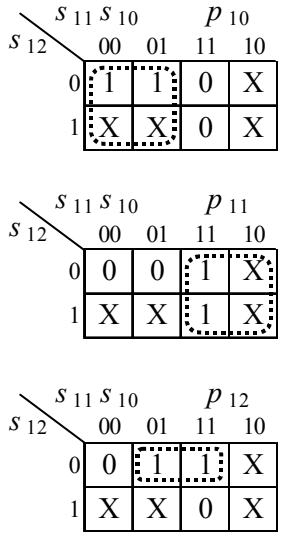

Figure 5.33 Logic minimization for the logic table in Figure 5.31.

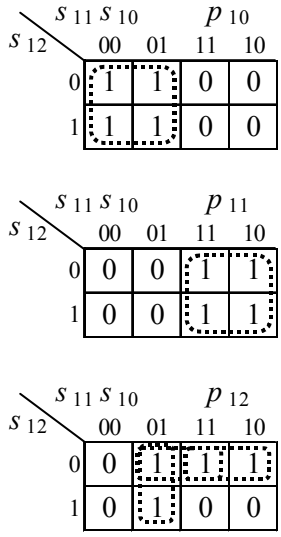

Figure 5.34 Logic minimization for the logic table in Figure 5.32.

Using the results of Figure 5.33 and Figure 5.34, the logic equations for the first MRS residue position bits *without* error correction are

$$\begin{aligned} p_{10} &= \overline{s_{11}}, \\ p_{11} &= s_{11}, \\ p_{12} &= s_{10}\overline{s_{12}}, \end{aligned} \tag{5.41}$$

and the logic equations for the first MRS residue position bits *with* error correction are

$$\begin{aligned} p_{10} &= \overline{s_{11}}, \\ p_{11} &= s_{11}, \\ p_{12} &= s_{10}\overline{s_{12}} + s_{11}\overline{s_{12}} + s_{10}\overline{s_{11}}. \end{aligned} \tag{5.42}$$

No error correction will be performed in this design since the thermometer code input is guaranteed to be error-free. The logic table for the second MRS residue position bits is given in Figure 5.35 and the logic minimization is provided in Figure 5.36.

2^{nd} MRS Residue

RSNS Residue				Position Bits			
s_{23}	s_{22}	s_{21}	s_{20}	p_{23}	p_{22}	p_{21}	p_{20}
0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1
0	0	1	0	X	X	X	X
0	0	1	1	1	0	1	0
0	1	0	0	X	X	X	X
0	1	0	1	X	X	X	X
0	1	1	0	X	X	X	X
0	1	1	1	0	1	1	0
1	0	0	0	X	X	X	X
1	0	0	1	X	X	X	X
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X

Figure 5.35 Logic table for computing the second MRS (s_2) position bits.

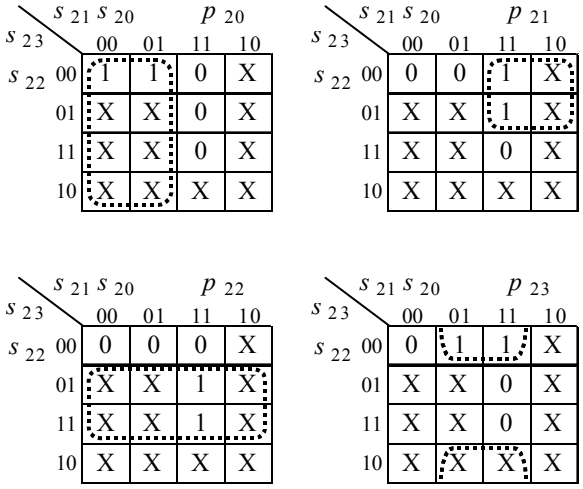

Figure 5.36 Logic minimization for the logic table in Figure 5.35.

Using Figure 5.36, the logic equations for the second MRS residue position bits are

$$\begin{aligned} p_{20} &= \overline{s_{21}}, \\ p_{21} &= s_{21}\overline{s_{23}}, \\ p_{22} &= s_{22}, \\ p_{23} &= s_{20}\overline{s_{22}}. \end{aligned} \tag{5.43}$$

The logic table for the third MRS residue position bits is given in Figure 5.37 and the logic minimization is provided in Figure 5.38.

3rd MRS Residue										
RSNS Residue					Position Bits					
s_{34}	s_{33}	s_{32}	s_{31}	s_{30}	p_{34}	p_{33}	p_{32}	p_{31}	p_{30}
0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	X	X	X	X	X
0	0	0	1	1	1	0	0	1	0
0	0	1	0	0	X	X	X	X	X
0	0	1	0	1	X	X	X	X	X
0	0	1	1	0	X	X	X	X	X
0	0	1	1	1	0	1	0	1	0
0	1	0	0	0	X	X	X	X	X
0	1	0	0	1	X	X	X	X	X
0	1	0	1	0	X	X	X	X	X
0	1	0	1	1	X	X	X	X	X
0	1	1	0	0	X	X	X	X	X
0	1	1	0	1	X	X	X	X	X
0	1	1	1	0	X	X	X	X	X
0	1	1	1	1	0	1	1	0	0
1	0	0	0	0	X	X	X	X	X
1	0	0	0	1	X	X	X	X	X
1	0	0	1	0	X	X	X	X	X
1	0	0	1	1	X	X	X	X	X
1	0	1	0	0	X	X	X	X	X
1	0	1	0	1	X	X	X	X	X
1	0	1	1	0	X	X	X	X	X
1	0	1	1	1	X	X	X	X	X
1	1	0	0	0	X	X	X	X	X
1	1	0	0	1	X	X	X	X	X
1	1	0	1	0	X	X	X	X	X
1	1	0	1	1	X	X	X	X	X
1	1	1	0	0	X	X	X	X	X
1	1	1	0	1	X	X	X	X	X
1	1	1	1	0	X	X	X	X	X

Figure 5.37 Logic table for computing the third MRS (s_3) position bits.

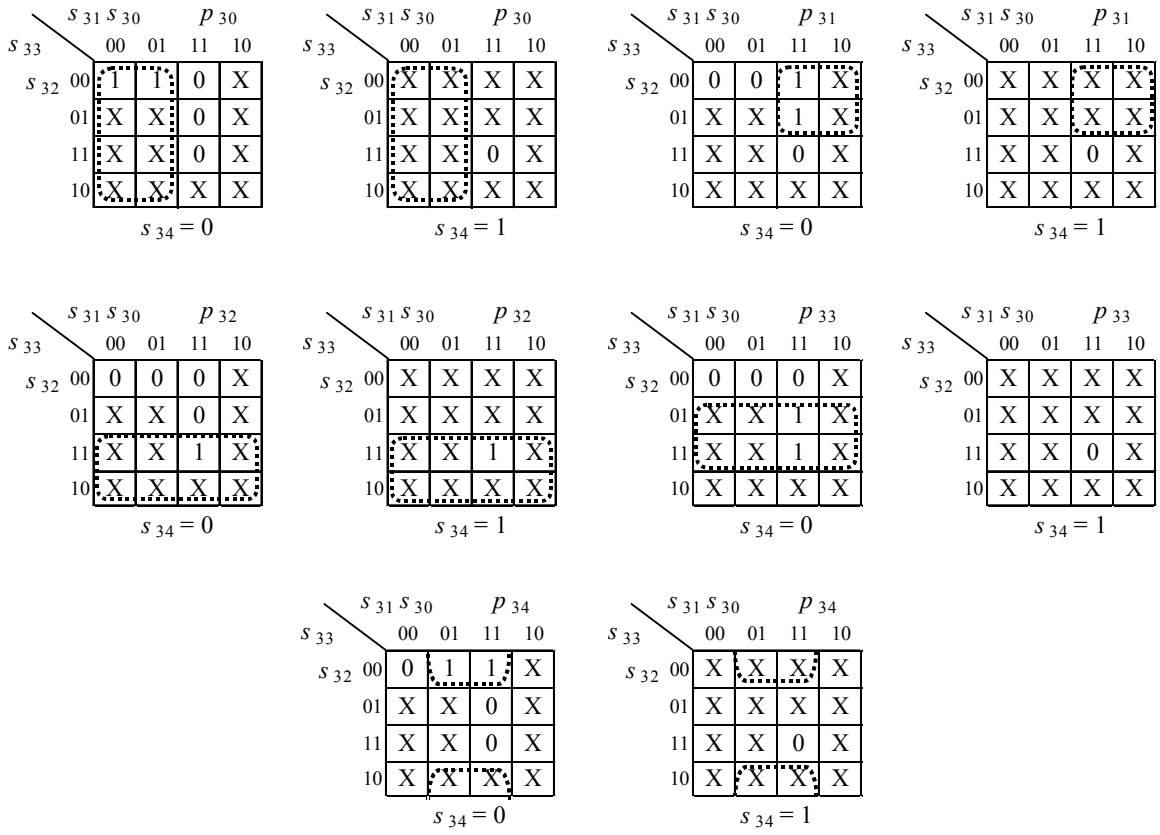

Figure 5.38 Logic minimization for the logic table in Figure 5.37.

Using Figure 5.38, the logic equations for the third MRS residue position bits are

$$\begin{aligned} p_{30} &= \overline{s_{31}}, \\ p_{31} &= s_{31}\overline{s_{33}}, \\ p_{32} &= s_{33}, \\ p_{33} &= s_{32}\overline{s_{34}}, \\ p_{34} &= s_{30}\overline{s_{32}}. \end{aligned} \tag{5.44}$$

The logic table used to compute the even residue flags for each MRS (s_1, s_2, and s_3) is shown in Figure 5.39 and the logic minimization is provided in Figure 5.40. The term *flag* is used in this instance to signify a single bit condition indicator. That is, the even residue flags are single bit indicators that are asserted then the corresponding residue is even and negated when the corresponding residue is odd.

1st MRS Even Flag (e_1)			
s_{12}	s_{11}	s_{10}	e_1
0	0	0	1
0	0	1	0
0	1	0	X
0	1	1	1
1	0	0	X
1	0	1	X
1	1	0	X
1	1	1	0

2nd MRS Even Flag (e_2)				
s_{23}	s_{22}	s_{21}	s_{20}	e_2
0	0	0	0	1
0	0	0	1	0
0	0	1	0	X
0	0	1	1	1
0	1	0	0	X
0	1	0	1	X
0	1	1	0	X
0	1	1	1	0
1	0	0	0	X
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	1

3rd MRS Even Flag (e_3)					
s_{34}	s_{33}	s_{32}	s_{31}	s_{30}	e_3
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	X
0	0	0	1	1	1
0	0	1	0	0	X
0	0	1	0	1	X
0	0	1	1	0	X
0	0	1	1	1	0
0	1	0	0	0	X
0	1	0	0	1	X
0	1	0	1	0	X
0	1	0	1	1	X
0	1	1	0	0	X
0	1	1	0	1	X
0	1	1	1	0	X
0	1	1	1	1	1
1	0	0	0	0	X
1	0	0	0	1	X
1	0	0	1	0	X
1	0	0	1	1	X
1	0	1	0	0	X
1	0	1	0	1	X
1	0	1	1	0	X
1	0	1	1	1	X
1	1	0	0	0	X
1	1	0	0	1	X
1	1	0	1	0	X
1	1	0	1	1	X
1	1	1	0	0	X
1	1	1	0	1	X
1	1	1	1	0	X
1	1	1	1	1	0

Figure 5.39 Logic table for the even residue flags (e_1, e_2, e_3).

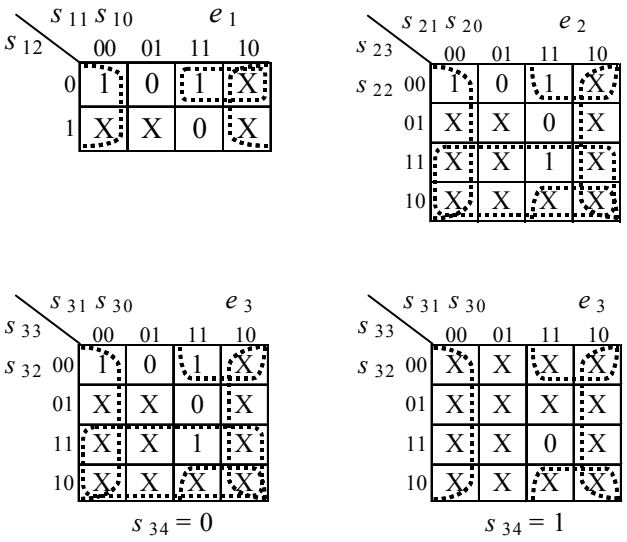

Figure 5.40 Logic minimization for the logic tables in Figure 5.39.

Using Figure 5.40, the logic equations for the even flags for each MRS are

$$\begin{aligned} e_1 &= \overline{s_{10}} + s_{11}\overline{s_{12}}, \\ e_2 &= s_{23} + \overline{s_{20}} + s_{21}\overline{s_{22}}, \\ e_3 &= s_{33}\overline{s_{34}} + \overline{s_{30}} + s_{31}\overline{s_{32}}. \end{aligned} \tag{5.45}$$

Finally, the logic table and logic minimization maps for the computation of the MRSS flags is provided in Figure 5.41. Using Figure 5.41, the logic equations for the three MRSS flags are

$$\begin{aligned} \overline{MRSS_0} &= e_3 \oplus e_1, \\ MRSS_1 &= e_3 \oplus e_2, \\ MRSS_2 &= e_2 \oplus e_1. \end{aligned} \tag{5.46}$$

In summary, the logic equations in this section compress the three thermometer-coded RSNS residues in the incoming vector X_h to position-bit-coded even 0th MRSS RNS residues. Furthermore, this section provides the logic equations for the F_{odd} and F_{MRSS} compensation factors. The next steps in the RSNS-to-binary conversion process as shown in Figure 5.18 are computing the LPS positional solution, index expansion, and index compensation.

e_3	e_2	e_1	MRSS Flags $MRSS_2$	$MRSS_1$	$MRSS_0$
0	0	0	0	0	1
0	0	1	1	0	0
0	1	0	X	X	X
0	1	1	0	1	0
1	0	0	0	1	0
1	0	1	X	X	X
1	1	0	1	0	0
1	1	1	0	0	1

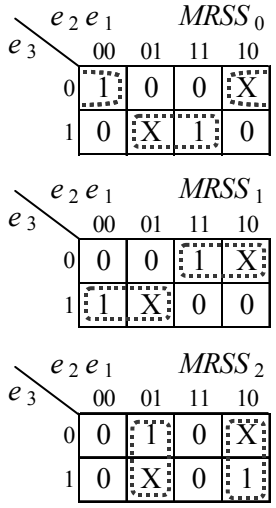

Figure 5.41 Logic tables and minimization for the MRSS flags ($MRSS_0$, $MRSS_1$, $MRSS_2$).

3. LPS Positional Solution and Index Expansion/Compensation

The LPS positional alignment method computes the least positive solution to the eight systems of congruence equations resulting from the application of (5.7) to the incoming RSNS vector residues. Aligning the RNS residue position bits computed in the previous section as shown in Figure 5.29 is accomplished in hardware by using eight 3-input NAND gates. Each NAND gate represents a column of position bits in Figure 5.29. When $\hat{M}$ starts on a 0^{th} MRSS boundary and ends on a 2^{nd} MRSS boundary, for each RSNS residue vector within $\hat{M}$, the output of only one NAND gate will be asserted. However, since this is not the case for this implementation, additional logic is necessary to adjust for the fact that $\hat{M}$ does not fit exactly within the solution range of the eight NAND gates. The additional logic ensures that any solution in the center six NAND gates has priority over a redundant solution that may appear in the first or eighth NAND gate. This additional logic would not be necessary if $\hat{M}$ were truncated to start on a 0^{th} MRSS boundary and end on a 2^{nd} MRSS boundary. The output of the eight NAND gates is converted to three binary bits using an 8-to-3 encoder.

The left circular shift of the position bits that relocate the $\hat{M}$ lower bound to zero is accomplished by wiring the position bits to the appropriate NAND gate. For instance, referring to the position bit shift in Figure 5.29, the position bits are not wired to the NAND gates in numerical order. Instead, position bits are rearranged as shown in the figure, and then wired to the NAND gate bank in the order shown. No logic gates are required to perform the shift. This step is made even clearer by the schematics provided in the following section.

The final step in the process is the conversion from the least positive solution to the binary representation of the position of the RSNS vector within the bounds of $\hat{M}$. The output of the NAND gates is the three-bit value $g/2$, so a left shift (implemented with wiring) converts the least positive solution to the four-bit index g. Following (5.35), $g + F_{odd}$ is computed next. The factor $F_{odd} = \overline{e}_1$ since e_1 is asserted when the first MRS residue (s_1) is even and therefore its complement is asserted when the first MRS residue (s_1) is odd. Furthermore, since g is a four-bit left-shifted version of the three-bit $g/2$, the least significant bit (LSB) of g is guaranteed to be zero. Therefore, $\overline{e}_1$ can replace the LSB of g with no additional logic required. Next, (5.35) requires the computation of $3(g + \overline{e}_1)$. Since a multiplier is a complex hardware function, it is easier to implement $3X = 2X + X$ in hardware using a wired shift and an adder. To accomplish this function, $g + \overline{e}_1$ is shifted left one position to form $2(g + \overline{e}_1)$ and used as the inputs to a six-bit adder. The concept is illustrated in Figure 5.42, where the notation g_b refers to the b^{th} bit of the binary representation of index g.

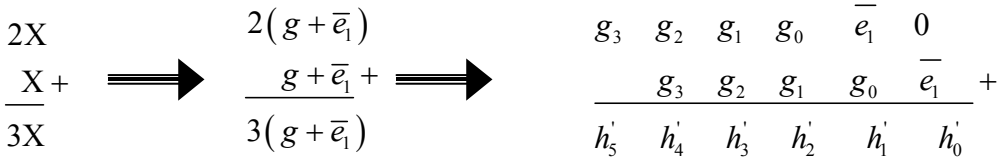

Figure 5.42 Using an adder to implement multiplication by three.

Finally, (5.35) requires the addition of the F_{MRSS} compensation factor. The F_{MRSS} compensation factor is represented by the signals $\overline{MRSS_0}$ and $MRSS_2$ from (5.46), which are the signals asserted if the residue vector is from the 1st or 2nd MRSS, respectively. Fortunately, the left shift of $g + \overline{e}_1$ provides a free LSB slot and the carry-in to the adder provides another free LSB slot. This permits $g + \overline{e}_1$, $2(g + \overline{e}_1)$, $\overline{MRSS_0}$, and $MRSS_2$ to be summed in a single adder as shown in Figure 5.43.

g_3	g_2	g_1	g_0	$\overline{e_1}$	$\overline{MRSS_0}$	$\leftarrow MRSS_2$ carry-in
	g_3	g_2	g_1	g_0	$\overline{e_1}$	+
h_5	h_4	h_3	h_2	h_1	h_0	

Figure 5.43 Addition for converting least positive solution to binary.

The output of the adder is the binary representation of the position *h* within $\hat{M}$ for the vector $X_h = [s_1 \quad s_2 \quad s_3]^T$. The next section provides the circuit descriptions and schematics that implement the logic equations developed above.

4. RSNS-to-Binary Circuit Schematics

Figure 5.44 is a schematic of the hardware implementation of the RSNS-to-binary conversion process for moduli $[m_1 \quad m_2 \quad m_3]^T = [3 \quad 4 \quad 5]^T$. The dashed lines in the figure indicate the portions of the schematic that are shown enlarged in subsequent figures.

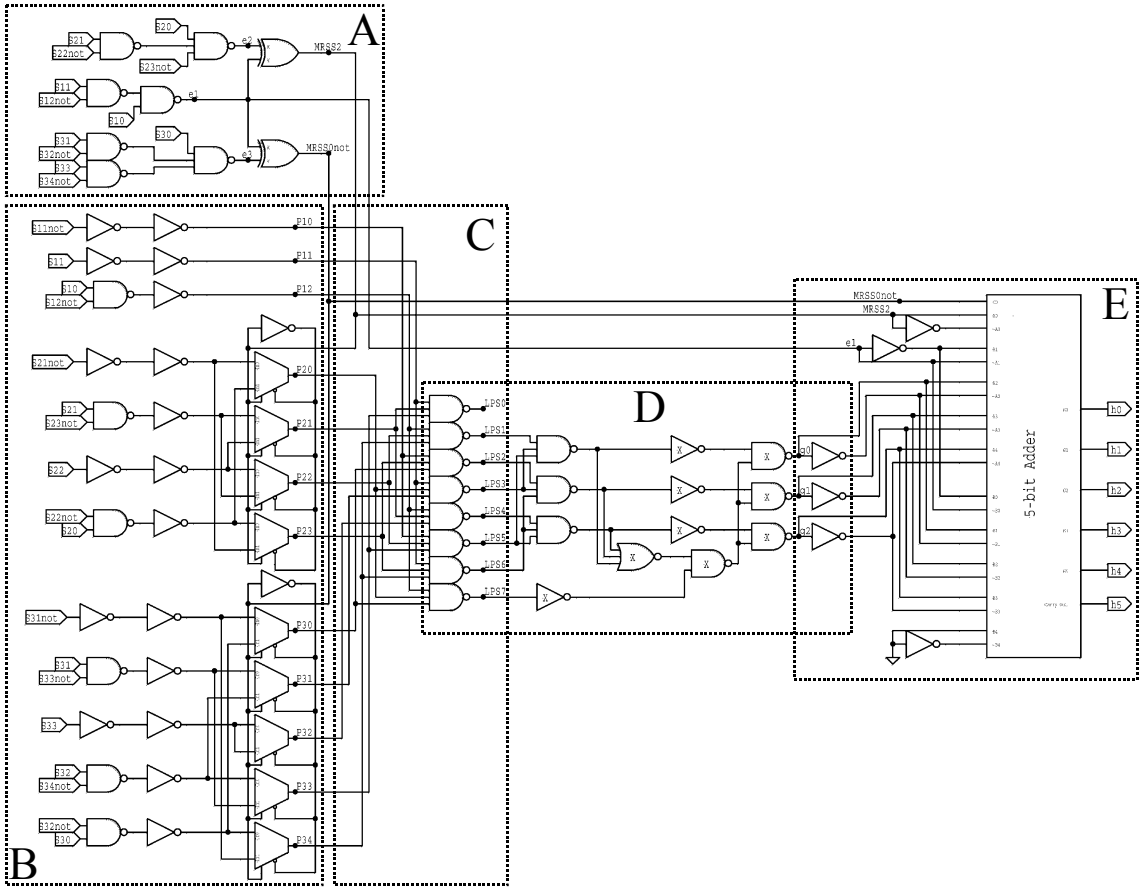

Figure 5.44 RSNS-to-binary conversion circuit for moduli $[m_1 \quad m_2 \quad m_3]^{\mathrm{T}} = [3 \quad 4 \quad 5]^{\mathrm{T}}$.

Figure 5.45 shows the portion of the design that applies (5.45) and (5.46) to generate e_1, e_2, e_3, $\overline{MRSS_0}$, and $MRSS_2$ from the RSNS residue thermometer code.

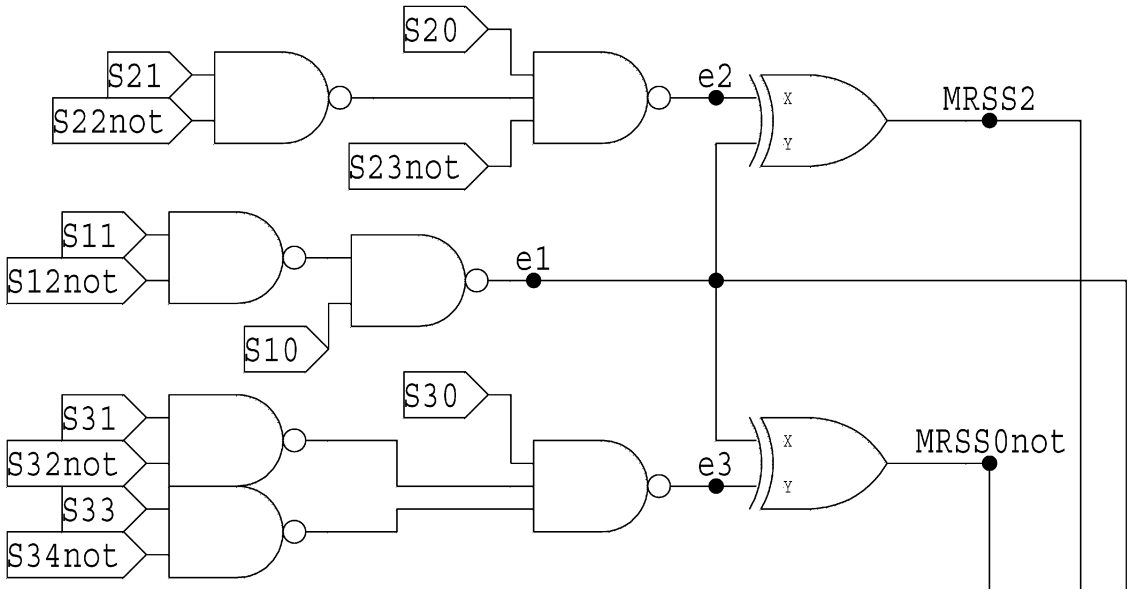

Figure 5.45 Circuit for generating e_1, e_2, e_3, $\overline{MRSS_0}$, and $MRSS_2$ (Figure 5.44 A).

Figure 5.46 shows the portion of the design that applies (5.41)-(5.44) to convert the RSNS residue thermometer code to RNS residue position bits and select normal or reversed position bits based on the residue vector MRSS. Nine two-to-one multiplexers using the MRSS signals $\overline{MRSS_0}$ and $MRSS_2$ as select inputs provide the mechanism for generating normal or reversed position bits. Notice that only the position bits for the second and third residue MRS require multiplexers as indicated in (5.7), (5.8), and (5.9).

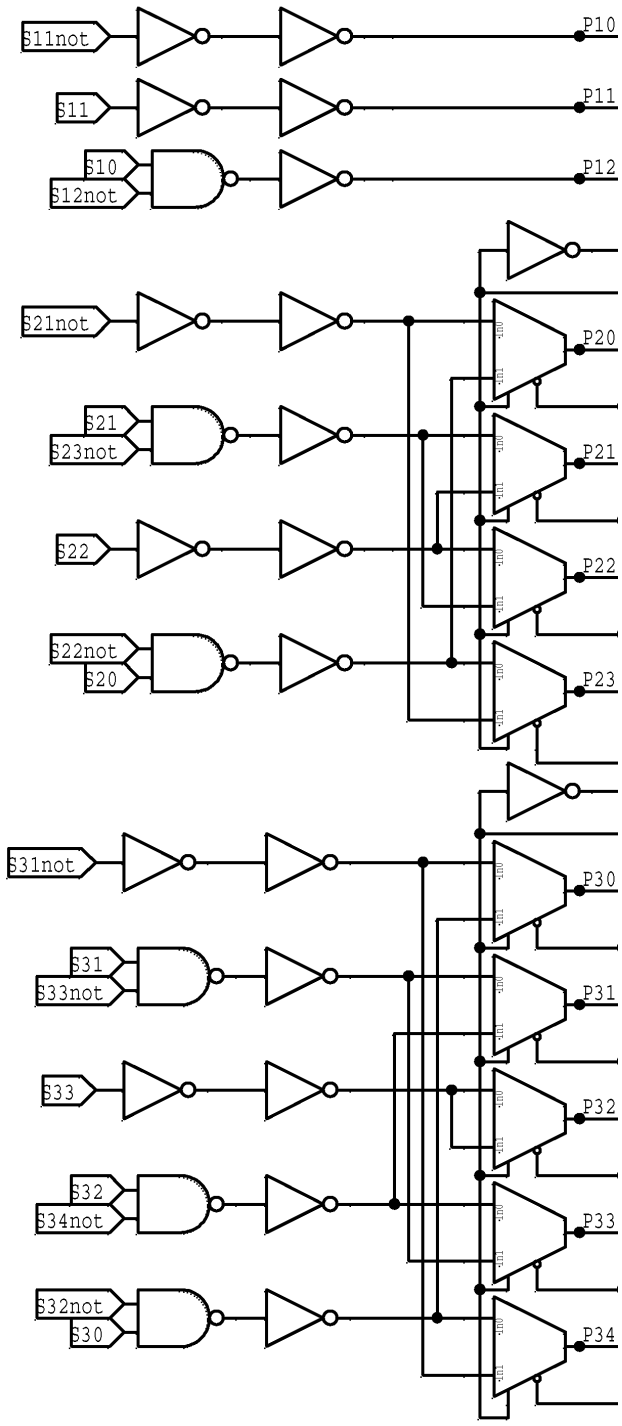

Figure 5.46 Circuit selecting normal or reversed position bits (Figure 5.44 B).

Figure 5.47 highlights the portion of the converter design that connects the position bits to the bank of eight LPS NAND gates.

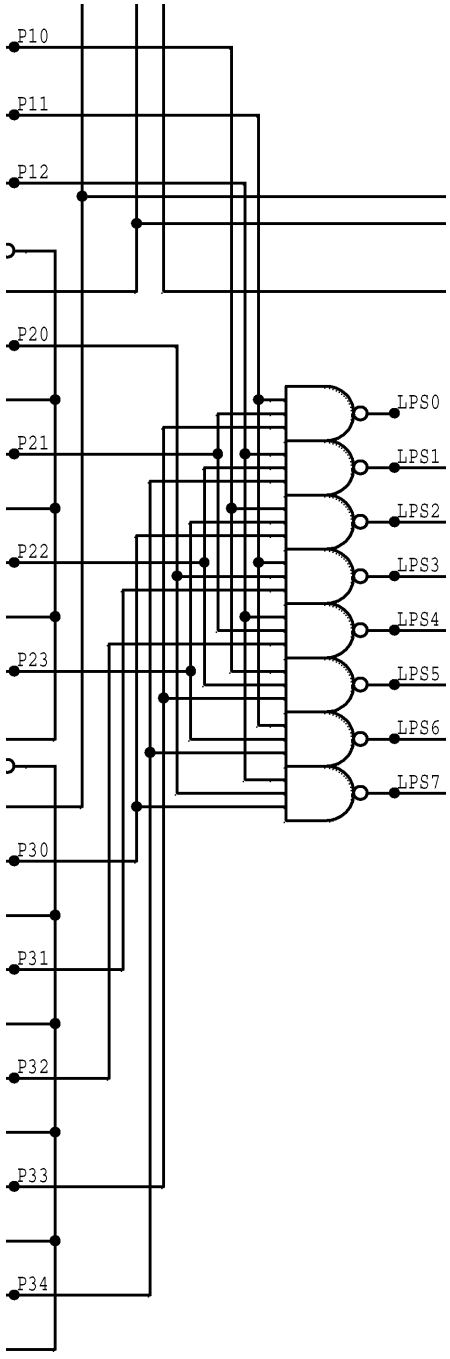

Figure 5.47 Position bit connections to the LPS NAND gate bank (Figure 5.44 C).

Each NAND gate represents a column in Figure 5.29 and consequently each gate has as many inputs as the columns have elements. The shift of the position bits to relocate the $\hat{M}$ lower bound to an index of zero is accomplished by wiring the position bits to the appropriate NAND gate. Figure 5.48 shows the portion of the schematic that con-

verts the least positive solution from the bank of NAND gates to the three-bit $g/2$ position.

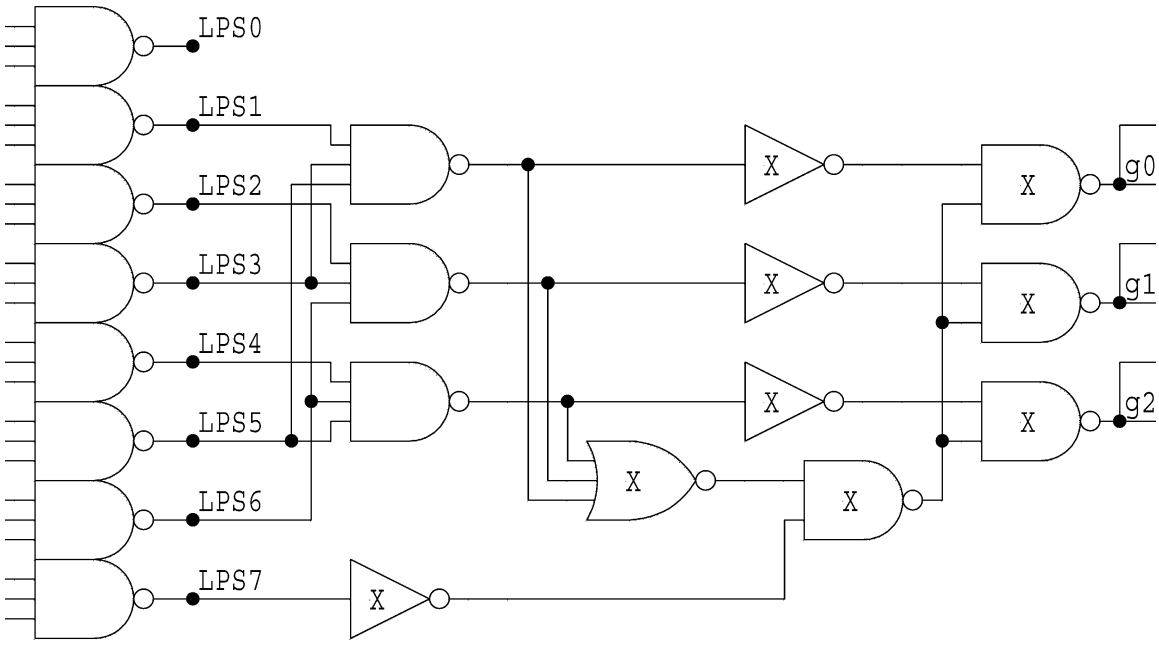

Figure 5.48 Positional least positive solution of congruence equations (Figure 5.44 D).

The output of the eight NAND gates is converted to three binary bits using an 8-to-3 encoder. The additional NAND, NOR, and inverter gates marked with an X in the figure ensure that any solution in the center six NAND gates has priority over a redundant solution that may appear in the first or last NAND gate. Redundancies can occur in this scheme because $\hat{M}$ in this case was not truncated to begin on a 0^{th} MRSS boundary and end on a 2^{nd} MRSS boundary. If $\hat{M}$ was truncated appropriately, the additional gates would not be necessary and the circuit would be much simpler. The size of the $\hat{M}$ would only be 36 rather than 43 and only 6 NAND gates would be required to compute the least positive positional solution. Figure 5.49 shows the portion of the circuit that converts the three-bit index $g/2$ to the six-bit index h using a five-bit carry look-ahead adder with carry-out.

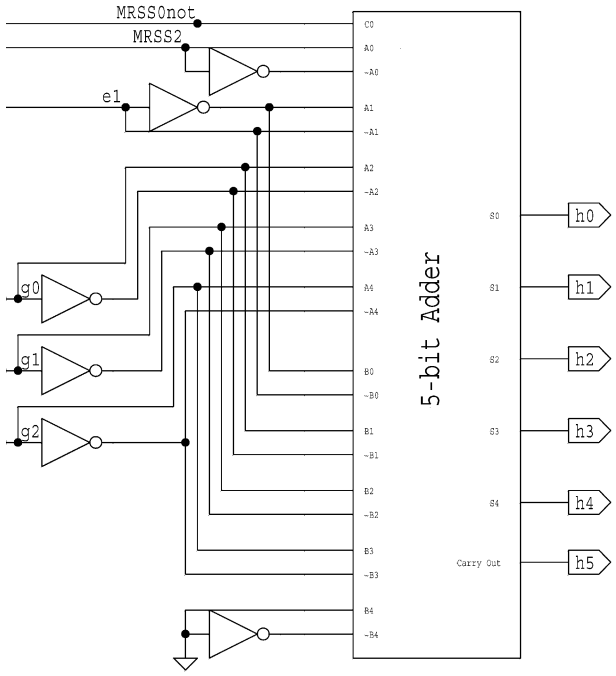

Figure 5.49 Converting $g/2$ to h using a six-bit carry look-ahead adder (Figure 5.44 E).

Using the logic equations from the previous section, the LPS RSNS-to-binary circuit was relatively straightforward to construct and required only 550 transistors, which is less than a third of the size of the decoder and one-tenth the size of the ROM (based on transistor count). Furthermore, this circuit can be pipelined with as few as six pipeline registers to achieve fast operation speeds. The techniques used to form the three-modulus LPS RSNS-to-binary converter can be extended to the general *N*-modulus converter case. The next section presents a procedure for the design of compact, high-speed *N*-modulus RSNS-to-binary converters.

D. *N*-MODULUS RSNS-TO-BINARY LPS CONVERSION

The *N*-modulus RSNS-to-binary conversion follows the same procedure as the three-modulus RSNS-to-binary conversion process detailed in Figure 5.18. The *N*-modulus form of (5.35) is

$$h = N(g + F_{odd}) + F_{MRSS}, \tag{5.47}$$

where

$$F_{odd} = \begin{Bmatrix} 0 & \text{if } s_1 \text{ is even} \\ 1 & \text{if } s_1 \text{ is odd} \end{Bmatrix},$$

$$F_{MRSS} = \begin{Bmatrix} 0 & \text{if } 0^{\text{th}} \text{ MRSS} \\ 1 & \text{if } 1^{\text{st}} \text{ MRSS} \\ \vdots & \vdots \\ N-1 & \text{if } (N\text{-}1)^{\text{st}} \text{ MRSS} \end{Bmatrix}. \tag{5.48}$$

The index g for the N-modulus case is computed in a similar manner to the three-modulus case. The first step is the computation of the RNS position bits from the RSNS thermometer code bits. Recall that the logic equations for the three-modulus position bit computation from (5.41), (5.43), and (5.44) are

$$\begin{aligned} p_{10} &= \overline{s_{11}}, \\ p_{11} &= s_{11}, \\ p_{12} &= s_{10}\overline{s_{12}}, \end{aligned} \tag{5.49}$$

for a modulus of three,

$$\begin{aligned} p_{20} &= \overline{s_{21}}, \\ p_{21} &= s_{21}\overline{s_{23}}, \\ p_{22} &= s_{22}, \\ p_{23} &= s_{20}\overline{s_{22}}, \end{aligned} \tag{5.50}$$

for a modulus of four, and

$$\begin{aligned} p_{30} &= \overline{s_{31}}, \\ p_{31} &= s_{31}\overline{s_{33}}, \\ p_{32} &= s_{33}, \\ p_{33} &= s_{32}\overline{s_{34}}, \\ p_{34} &= s_{30}\overline{s_{32}}, \end{aligned} \tag{5.51}$$

for a modulus of five. Extending these examples to the general case results in the RNS position bit logic equations

$$
\begin{aligned}
p_{i0} &= \overline{s_{i1}}, \\
p_{i1} &= s_{i1}\overline{s_{i3}}, \\
p_{i2} &= s_{i3}\overline{s_{i5}}, \\
p_{i3} &= s_{i5}\overline{s_{i7}}, \\
&\vdots \\
p_{i\left(\frac{m_i}{2}-1\right)} &= s_{i(m_i-3)}\overline{s_{i(m_i-1)}}, \\
p_{i\left(\frac{m_i}{2}\right)} &= s_{i(m_i-2)}, \\
p_{i\left(\frac{m_i}{2}+1\right)} &= s_{i(m_i-4)}\overline{s_{i(m_i-2)}}, \\
&\vdots \\
p_{i(m_i-3)} &= s_{i6}\overline{s_{i8}}, \\
p_{i(m_i-2)} &= s_{i4}\overline{s_{i6}}, \\
p_{i(m_i-1)} &= s_{i2}\overline{s_{i4}}, \\
p_{i(m_i-1)} &= s_{i0}\overline{s_{i2}},
\end{aligned}
\tag{5.52}
$$

when the modulus is even, and results in the RNS position bit logic equations

$$
\begin{aligned}
p_{i0} &= \overline{s_{i1}}, \\
p_{i1} &= s_{i1}\overline{s_{i3}}, \\
p_{i2} &= s_{i3}\overline{s_{i5}}, \\
p_{i3} &= s_{i5}\overline{s_{i7}}, \\
&\vdots \\
p_{i\left(\left\lfloor \frac{m_i}{2} \right\rfloor -1\right)} &= s_{i(m_i-4)}\overline{s_{i(m_i-2)}}, \\
p_{i\left(\left\lfloor \frac{m_i}{2} \right\rfloor\right)} &= s_{i(m_i-1)}, \\
p_{i\left(\left\lfloor \frac{m_i}{2} \right\rfloor +1\right)} &= s_{i(m_i-3)}\overline{s_{i(m_i-1)}}, \\
&\vdots \\
p_{i(m_i-3)} &= s_{i6}\overline{s_{i8}}, \\
p_{i(m_i-2)} &= s_{i4}\overline{s_{i6}}, \\
p_{i(m_i-1)} &= s_{i2}\overline{s_{i4}}, \\
p_{i(m_i-1)} &= s_{i0}\overline{s_{i2}},
\end{aligned}
\tag{5.53}
$$

when the modulus is odd. The variable m_i is the modulus and the subscript i is the MRS number.

The conditional position bit reversal is accomplished in the same manner as the three-modulus case. The position bits in all MRSs except the first MRS are wired to 2-to-1 multiplexers, with the bits ordered according to (5.52) and (5.53) connected to one input of the multiplexer and the reversed bits connected to the other multiplexer input. The multiplexer select is controlled by the MRSS flags as in the three-modulus case.

Like the three-modulus case, the N-modulus LPS RSNS-to-binary conversion requires computation of the even residue flags and the MRSS residue flags. The logic equations for the three-modulus even residue flags from (5.45) are

$$
\begin{aligned}
e_1 &= \overline{s_{10}} + s_{11}\overline{s_{12}}, \\
e_2 &= \overline{s_{20}} + s_{21}\overline{s_{22}} + s_{23}, \\
e_3 &= \overline{s_{30}} + s_{31}\overline{s_{32}} + s_{33}\overline{s_{34}}.
\end{aligned} \tag{5.54}
$$

Extending these examples to the general *N*-modulus case results in

$$
e_i = \overline{s_{i0}} + s_{i1}\overline{s_{i2}} + s_{i3}\overline{s_{i4}} + \cdots + s_{i(m_i-1)}, \tag{5.55}
$$

for an even modulus, and

$$
e_i = \overline{s_{i0}} + s_{i1}\overline{s_{i2}} + s_{i3}\overline{s_{i4}} + \cdots + s_{i(m_i-2)}\overline{s_{i(m_i-1)}}, \tag{5.56}
$$

for an odd modulus. Similarly, the logic equations for the three-modulus MRSS flags from (5.45) are

$$
\begin{aligned}
\overline{MRSS_0} &= e_3 \oplus e_1, \\
MRSS_1 &= e_3 \oplus e_2, \\
MRSS_2 &= e_2 \oplus e_1.
\end{aligned} \tag{5.57}
$$

Extending these examples to the *N*-modulus case produces

$$
\begin{aligned}
MRSS_{N-1} &= e_1 \oplus e_2, \\
MRSS_{N-2} &= e_2 \oplus e_3, \\
&\vdots \\
MRSS_2 &= e_{N-2} \oplus \overline{e_{N-1}}, \\
MRSS_1 &= e_{N-1} \oplus \overline{e_N}, \\
MRSS_0 &= e_N \oplus \overline{e_1}.
\end{aligned} \tag{5.58}
$$

Like the three-modulus example, the computation of the *N*-modulus least positive solution is performed using a bank of NAND gates. The index compression factor was determined in the previous section to be 2*N* the number of *N*-input NAND gates required in the LPS bank was given by (5.30) where $\hat{M}$ is the longest sequence of unique RSNS vectors in the fundamental period. For the *N*-modulus case, $\hat{M}$ is assumed to be truncated such that it begins on a 0^{th} MRSS boundary and ends on an $(N-1)^{\text{th}}$ MRSS bound-

ary. This means that $\hat{M}$ will be an even multiple of $2N$ and there will be no redundancy problems at the extreme ends of the NAND gate LPS bank.

Correctly connecting the position bits to the LPS NAND gate bank shifts the lower bound of $\hat{M}$ to the first NAND gate. Consequently, all $\hat{M}$ vector positions are contained within the number of NAND gates given by (5.30). The position bits in each MRS computed by (5.52) or (5.53) must be left shifted by an amount equal to the corresponding RNS residue in the vector $XR_{h_{start}}$, where h_{start} is the index of the 0^{th} MRSS vector greater or equal to the lower bound of $\hat{M}$.

Since the output of only one LPS NAND gate is asserted for each RSNS residue vector within $\hat{M}$, the output of the NAND bank is converted to a binary representation with a standard encoder. The output of the encoder is the index $g/2$. The number of bits required for the binary representation of the index $g/2$ at the output of the encoder is

$$b = \log_2\left(\frac{\hat{M}}{2N}\right). \tag{5.59}$$

Like the three-modulus case, the last step in the RSNS-to-binary conversion process is computing the unknown value *h* from the *b*-bit index $g/2$. This is accomplished by means of the relationship provided in (5.47), which represents the *expand* and *compensate* portion of the *compress-solve-expand-compensate* procedure described in Figure 5.18. First, the binary representation of $g/2$ is shifted left one position to form the index *g*. Next, since the compensation factor F_{odd} is always $\overline{e}_1$ as shown in the three-modulus converter, $\overline{e}_1$ becomes the LSB, forming $g + F_{odd}$. The next expansion, multiplying this result by *N*, is an interesting problem. For the three-modulus case, the trick $3\mathrm{X} = 2\mathrm{X} + \mathrm{X}$ was used so that a simple adder could be used in place of a multiplier. For a case where the number of MRSs is a power of two, the multiplication can be accomplished solely by wired shifts. For other cases, combinations of wired shifts and adders may be used. As an example, for the six-modulus case, $6\mathrm{X} = 4\mathrm{X} + 2\mathrm{X}$, where 4X and 2X may be formed by two-bit and one-bit wired shifts, respectively. As a last resort, a

multiplier may be used to compute $N(g + F_{odd})$. Finally, the MRSS compensation flag bits F_{MRSS} generated using (5.58) can be incorporated into the empty least significant bits, created by the left-shifts of the addend, and the carry-in of the adder as in the three-modulus case. However, if *N* is a power of two, the MRSS compensation flag bits are the least significant bits of the final solution and no adder or multiplier is required. Evidently, when *N* is a power of two, there are several advantages to RSNS-to-binary conversion using the LPS solution method. The next section implements a four-modulus RSNS-to-binary converter to illustrate an application of the theory presented in this section as well as decode the output of the four-channel ADC designed in Chapter III.

E. FOUR-MODULUS RSNS-TO-BINARY LPS CONVERTER

1. Logic Design

For the four-modulus RSNS-to-binary converter, the conversion equation from (5.47) is

$$h = 4(g + F_{odd}) + F_{MRSS}, \tag{5.60}$$

where

$$F_{odd} = \begin{Bmatrix} 0 & \text{if } s_1 \text{ is even} \\ 1 & \text{if } s_1 \text{ is odd} \end{Bmatrix},$$

$$F_{MRSS} = \begin{Bmatrix} 0 & \text{if } 0^{\text{th}} \text{ MRSS} \\ 1 & \text{if } 1^{\text{st}} \text{ MRSS} \\ 2 & \text{if } 2^{\text{nd}} \text{ MRSS} \\ 3 & \text{if } 3^{\text{rd}} \text{ MRSS} \end{Bmatrix}. \tag{5.61}$$

Since the four-modulus RSNS-to-binary converter designed in this section will decode the thermometer code bits from the output of the four-channel ADC in Chapter III, the design moduli must be identical. The moduli set used for the ADC and the converter was $\begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix} = \begin{bmatrix} 3 & 5 & 8 & 11 \end{bmatrix}$.

The position bit logic equations for the first MRS, modulus 3, and the second MRS, modulus 5, were given in (5.49) and (5.51). The third MRS has an even modulus, modulus 8, so the position bit logic equations using (5.52) are

$$\begin{aligned}
p_{30} &= \overline{s_{31}}, \\
p_{31} &= s_{31}\overline{s_{33}}, \\
p_{32} &= s_{33}\overline{s_{35}}, \\
p_{33} &= s_{35}\overline{s_{37}}, \\
p_{34} &= s_{36}, \\
p_{35} &= s_{34}\overline{s_{36}}, \\
p_{36} &= s_{32}\overline{s_{34}}, \\
p_{37} &= s_{30}\overline{s_{32}}.
\end{aligned} \tag{5.62}$$

The fourth MRS, modulus 11, is odd so (5.53) is applied and the position bit logic equations are

$$\begin{aligned}
p_{40} &= \overline{s_{41}}, \\
p_{41} &= s_{41}\overline{s_{43}}, \\
p_{42} &= s_{43}\overline{s_{45}}, \\
p_{43} &= s_{45}\overline{s_{47}}, \\
p_{44} &= s_{47}\overline{s_{49}}, \\
p_{45} &= s_{49}, \\
p_{46} &= s_{48}\overline{s_{410}}, \\
p_{47} &= s_{46}\overline{s_{48}}, \\
p_{48} &= s_{44}\overline{s_{46}}, \\
p_{49} &= s_{42}\overline{s_{44}}, \\
p_{410} &= s_{40}\overline{s_{42}}.
\end{aligned} \tag{5.63}$$

The conditional position bit reversal is accomplished in the same manner as the three-modulus case. The position bits in all MRSs except the first MRS are wired to 2-to-1 multiplexers, with the normal position bits connected to one input of the multiplexer and the reversed position bits connected to the other input.

From (5.55) and (5.56), the logic equations for the even residue flag computation for each MRS are

$$\begin{aligned} e_1 &= \overline{s_{10}} + s_{11}\overline{s_{12}}, \\ e_2 &= \overline{s_{20}} + s_{21}\overline{s_{22}} + s_{23}\overline{s_{24}}, \\ e_3 &= \overline{s_{30}} + s_{31}\overline{s_{32}} + s_{33}\overline{s_{34}} + s_{35}\overline{s_{36}} + s_{37}, \\ e_4 &= \overline{s_{40}} + s_{41}\overline{s_{42}} + s_{43}\overline{s_{44}} + s_{45}\overline{s_{46}} + s_{47}\overline{s_{48}} + s_{49}\overline{s_{410}}. \end{aligned} \tag{5.64}$$

From (5.58), the logic equations for the MRSS flags are

$$\begin{aligned} MRSS_3 &= e_1 \oplus e_2, \\ MRSS_2 &= e_2 \oplus e_3, \\ MRSS_1 &= e_3 \oplus e_4, \\ MRSS_0 &= e_4 \oplus \overline{e_1}. \end{aligned} \tag{5.65}$$

Using (5.30) and the fact that $\hat{M}$ for the 8-bit ADC was truncated to 256, the number of 4-input NAND gates required in the LPS solution bank is 32. Using the search algorithm from Chapter IV, the lower bound of the truncated $\hat{M}$ was found to be $h_{start} = 2616$, and the RNS vector was determined to be $XR_{2616} = \begin{bmatrix} 0 & 2 & 7 & 8 \end{bmatrix}^{\mathrm{T}}$. Therefore, left shifting the first, second, third, and fourth MRS position bits by zero, two, seven, and eight positions, respectively, before connecting them to the LPS NAND gate bank shifts the lower bound for $\hat{M}$ to the first NAND gate.

The 32-bit output of the LPS NAND gate bank is converted to a 5-bit binary value using a 32-to-8 encoder. The 5-bit output of the encoder is the index $g/2$. The relationship provided in (5.60) is used to convert $g/2$ to the unknown incoming value *h*. First, the 5-bit $g/2$ is shifted left one position and the compensation factor $F_{odd} = \overline{e}_1$ becomes the LSB to form $g + F_{odd}$. Multiplying this result by four is be accomplished by a two-bit wired shift since four is a power of two and therefore a special case in the LPS RSNS-to-binary conversion. Thus, the binary representation of the unknown incoming value *h* thus far is

$$\begin{bmatrix} h_7 & h_6 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \end{bmatrix} = \begin{bmatrix} g_4 & g_3 & g_2 & g_1 & g_0 & \overline{e}_1 & 0 & 0 \end{bmatrix}. \tag{5.66}$$

The two least significant bits represent the MRSS compensation flag bits F_{MRSS} and are the binary representation of the MRSS of the incoming residue vector X_h generated using (5.65). The F_{MRSS} bits corresponding to the RSNS vector MRSS are provided in Figure 5.50.

MRSS Compensation Bits

MRSS	$fMRSS_1$	$fMRSS_0$	$MRSS_3$	$MRSS_2$	$MRSS_1$	$MRSS_0$
0	0	0	0	0	0	1
1	0	1	0	0	1	0
2	1	0	0	1	0	0
3	1	1	1	0	0	0

Figure 5.50 MRSS compensation bits logic table.

Using Figure 5.50, the logic equations for the MRSS compensation are

$$\begin{aligned} fMRSS_0 &= MRSS_0 + MRSS_1, \\ fMRSS_1 &= MRSS_2 + MRSS_3, \end{aligned} \tag{5.67}$$

where $fMRSS_0$ and $fMRSS_1$ are the bit that make up the two-bit binary representation of the MRSS compensation factor F_{MRSS}. Therefore, the final binary representation of the position *h* for the thermometer-coded residue vector X_h is

$$\left[h_7\ h_6\ h_5\ h_4\ h_3\ h_2\ h_1\ h_0\right] = \left[g_4\ g_3\ g_2\ g_1\ g_0\ \overline{e}_1\ fMRSS_1\ fMRSS_0\right]. \tag{5.68}$$

Consequently, this section has provided all of the necessary logic equations and concepts to construct the circuits for the hardware implementation of the four-modulus LPS RSNS-to-binary converter.

2. RSNS-to-Binary Decoder Schematics

This section provides the circuit schematics for the four-modulus RSNS-to-binary converter for $\begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix} = \begin{bmatrix} 3 & 5 & 8 & 11 \end{bmatrix}$, based on the logic equations in the previous section. Figure 5.51 provides the schematic for the four-modulus RSNS-to-binary converter showing the position bit generation, LPS NAND gate bank, and associated $\hat{M}$ lower bound shift wiring. Figure 5.52 shows the details of the reverse position bit selectors for each MRS. Figure 5.53 provides a schematic of the 32-to-8 encoder. Finally, Figure 5.54 provides a schematic of the even residue flag and MRSS flag generation circuitry.

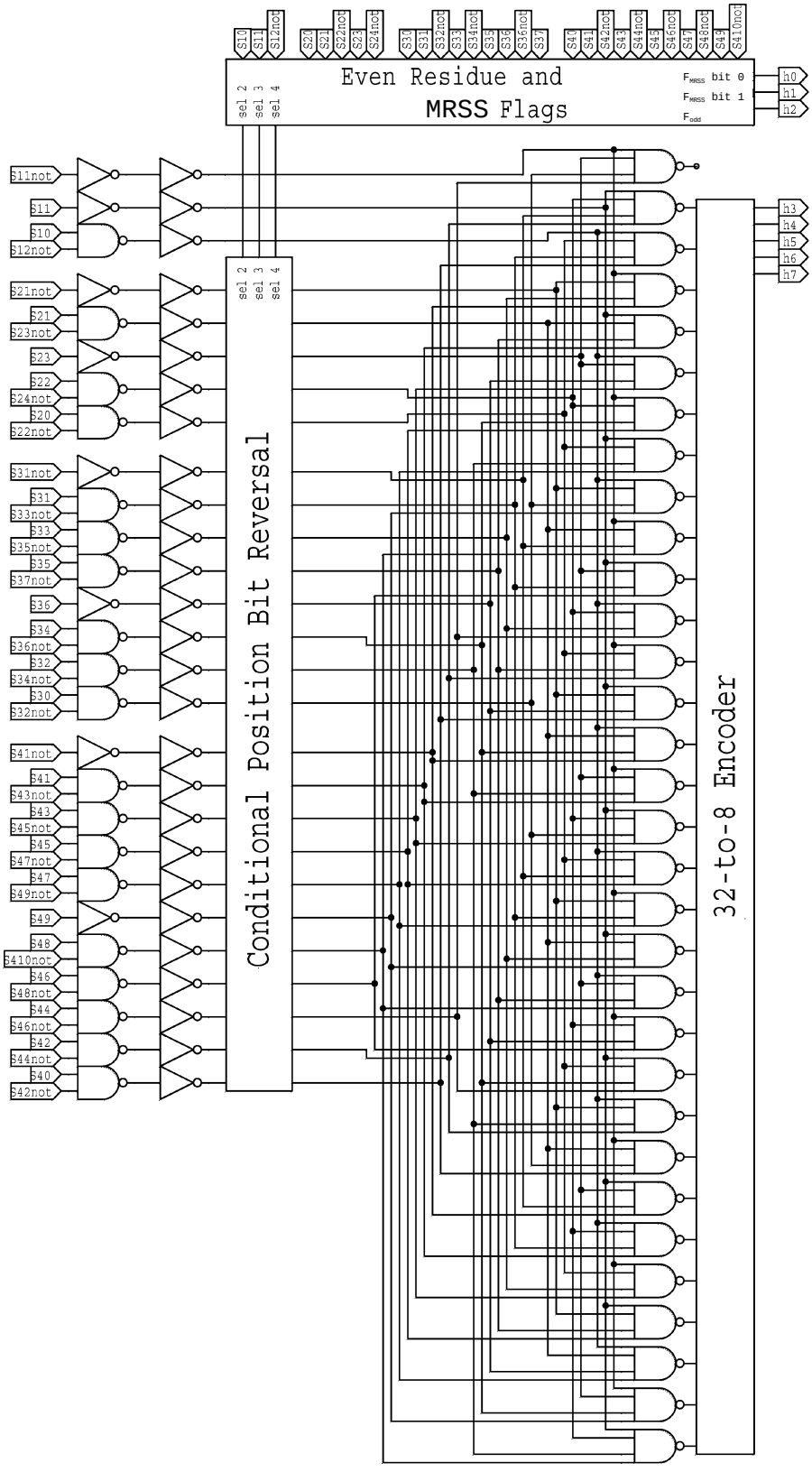

Figure 5.51 Four-modulus RSNS-to-binary converter schematic.

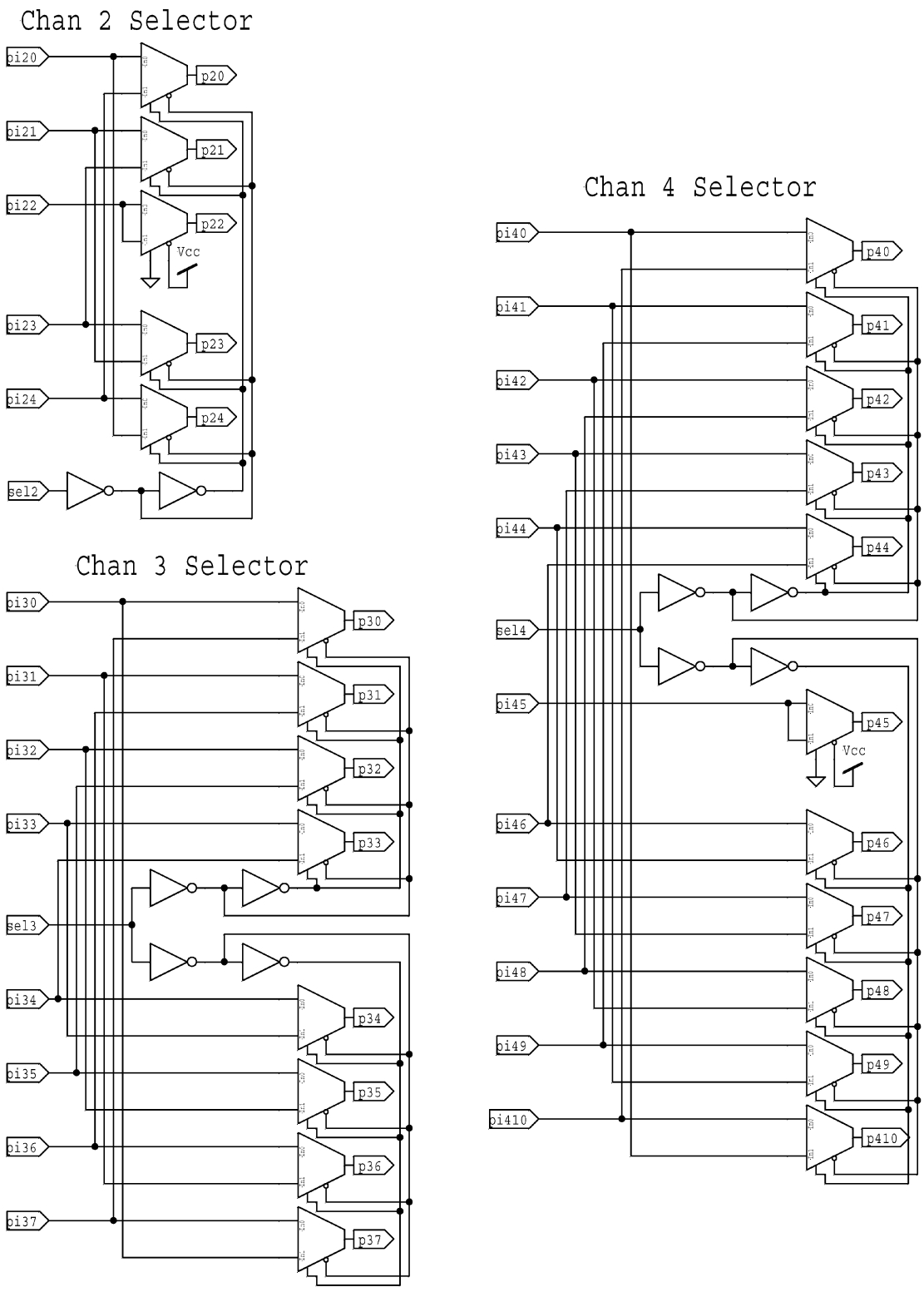

Figure 5.52 Reverse position bit selector circuit schematic.

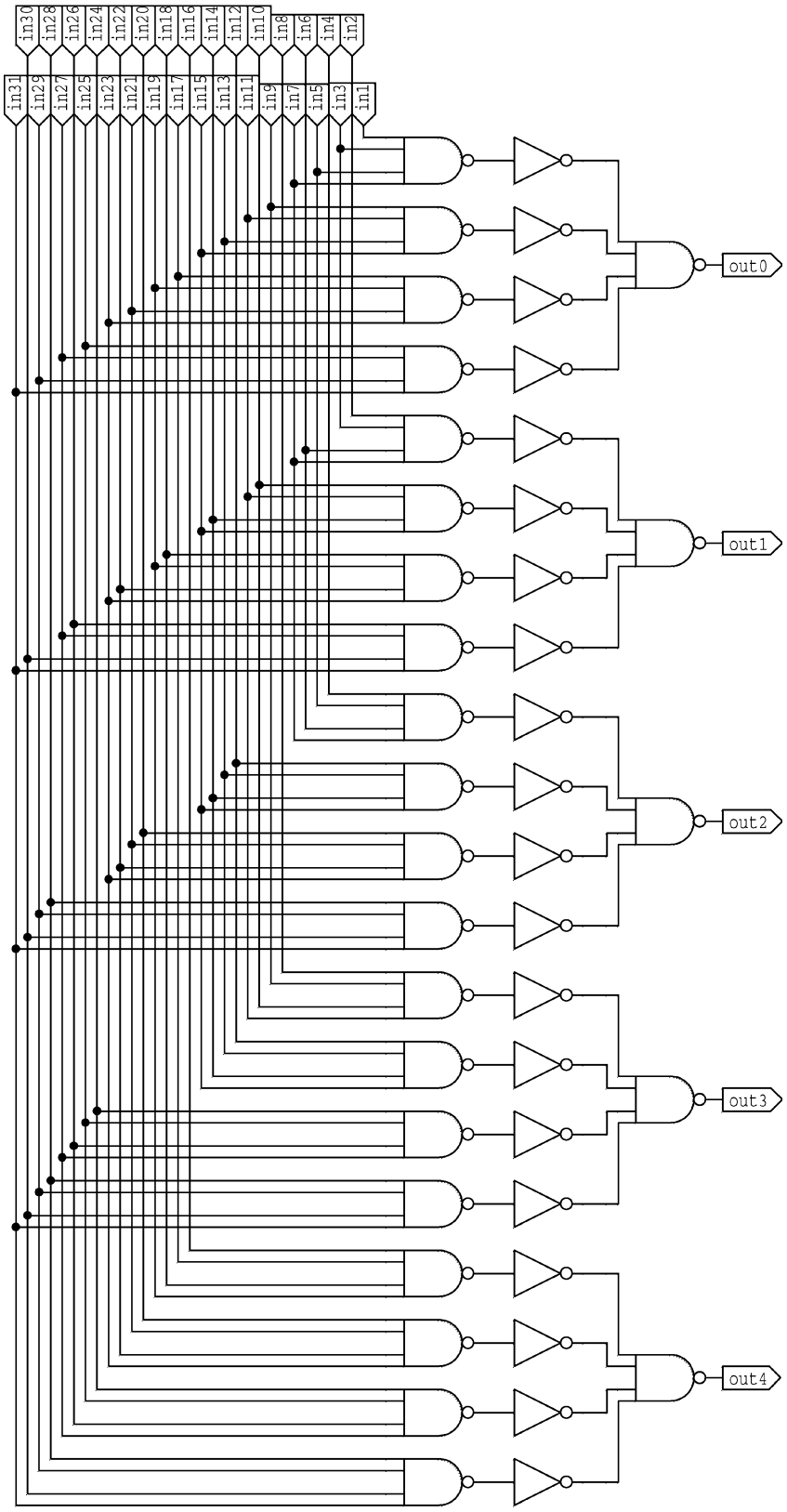

Figure 5.53 Schematic of the 32-to-8 encoder circuit.

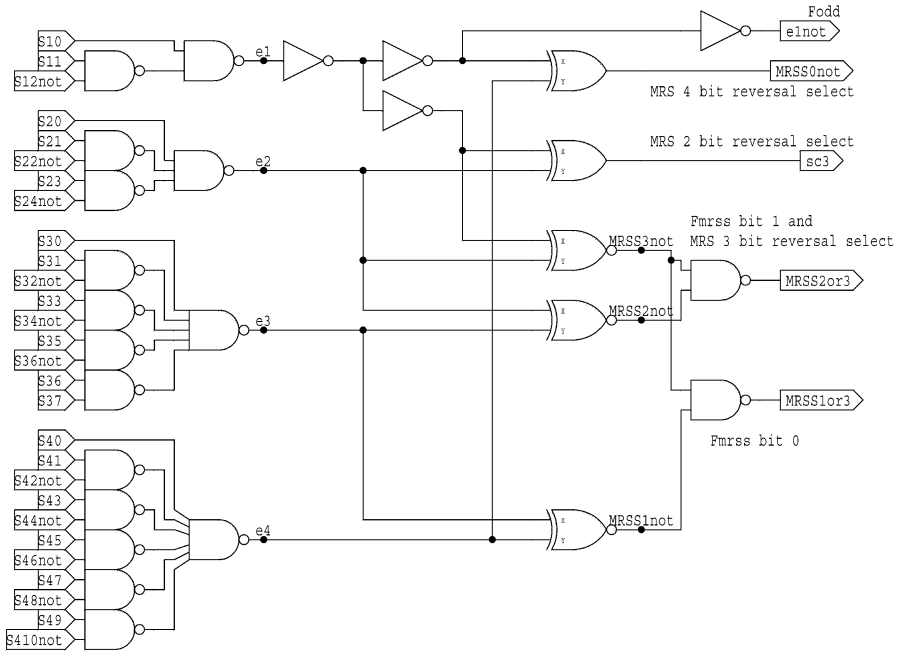

Figure 5.54 Even residue flag and MRSS flag generation circuit schematic.

In round numbers, the transistor count for this implementation was approximately 900 minimum-sized transistors. In contrast, conservative estimates for the ROM and decoder methods proposed at the beginning of this chapter are 295,000 transistors and 11,000 transistors, respectively. Consequently, the four-modulus RSNS-to-binary implementation is at least an order of magnitude smaller than the decoder and two orders of magnitude smaller than the ROM. Figure 5.55 shows a graph transistor count versus $\hat{M}$ for five representative four-modulus systems. The obvious trend is that the LPS RSNS-to-binary conversion method is consistently over 10 times smaller than the decode conversion method and over 100 times smaller than the ROM conversion method. Furthermore, the conversion implementation presented in this section is easily pipelined to produce high-speed circuit operation.

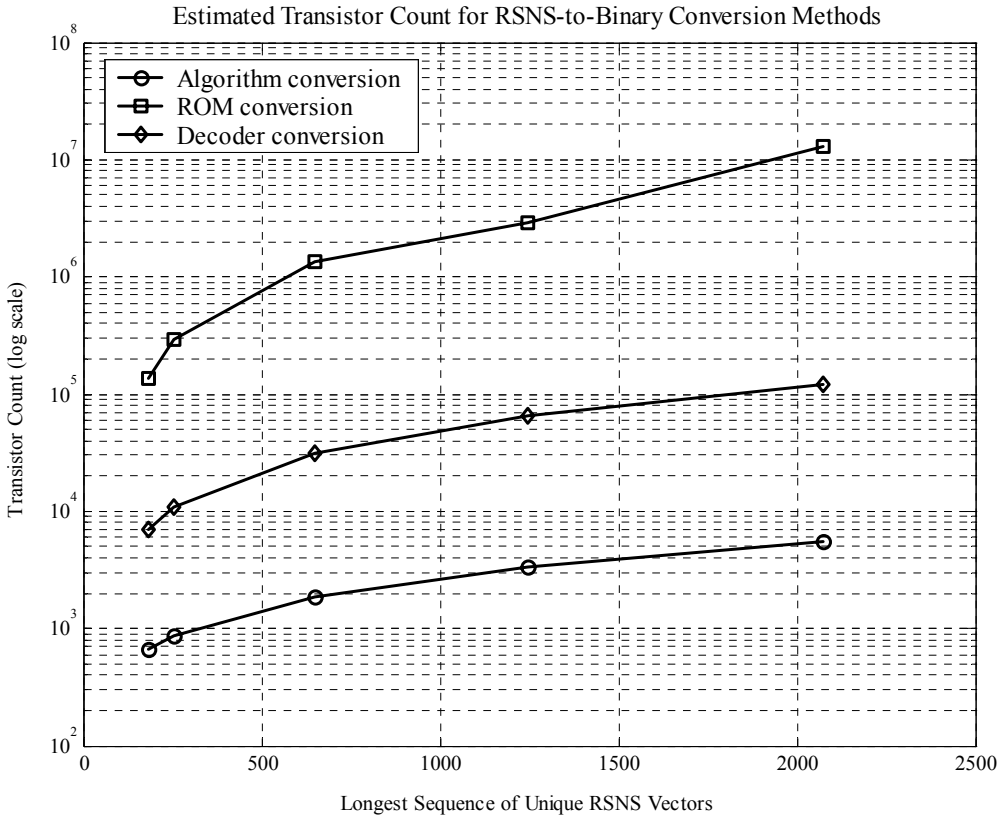

Figure 5.55 Transistor count comparison of three RSNS-to-binary conversion methods.

F. SIMULATION RESULTS

1. Three-modulus RSNS-to-Binary Conversion

The RSNS thermometer code output of the three-channel ADC from Chapter III was used to test the three-modulus RSNS-to-binary conversion circuit. Both the decoder and the LPS conversion circuits shown in Figure 5.1 and Figure 5.44 were connected to the ADC in parallel as shown in Figure 5.56 so that each conversion circuit received the same thermometer code signals from the ADC. The circuit was clocked at 200 MHz because neither conversion circuit was pipelined to match the 1-GHz speed used for testing the front end of the ADC.

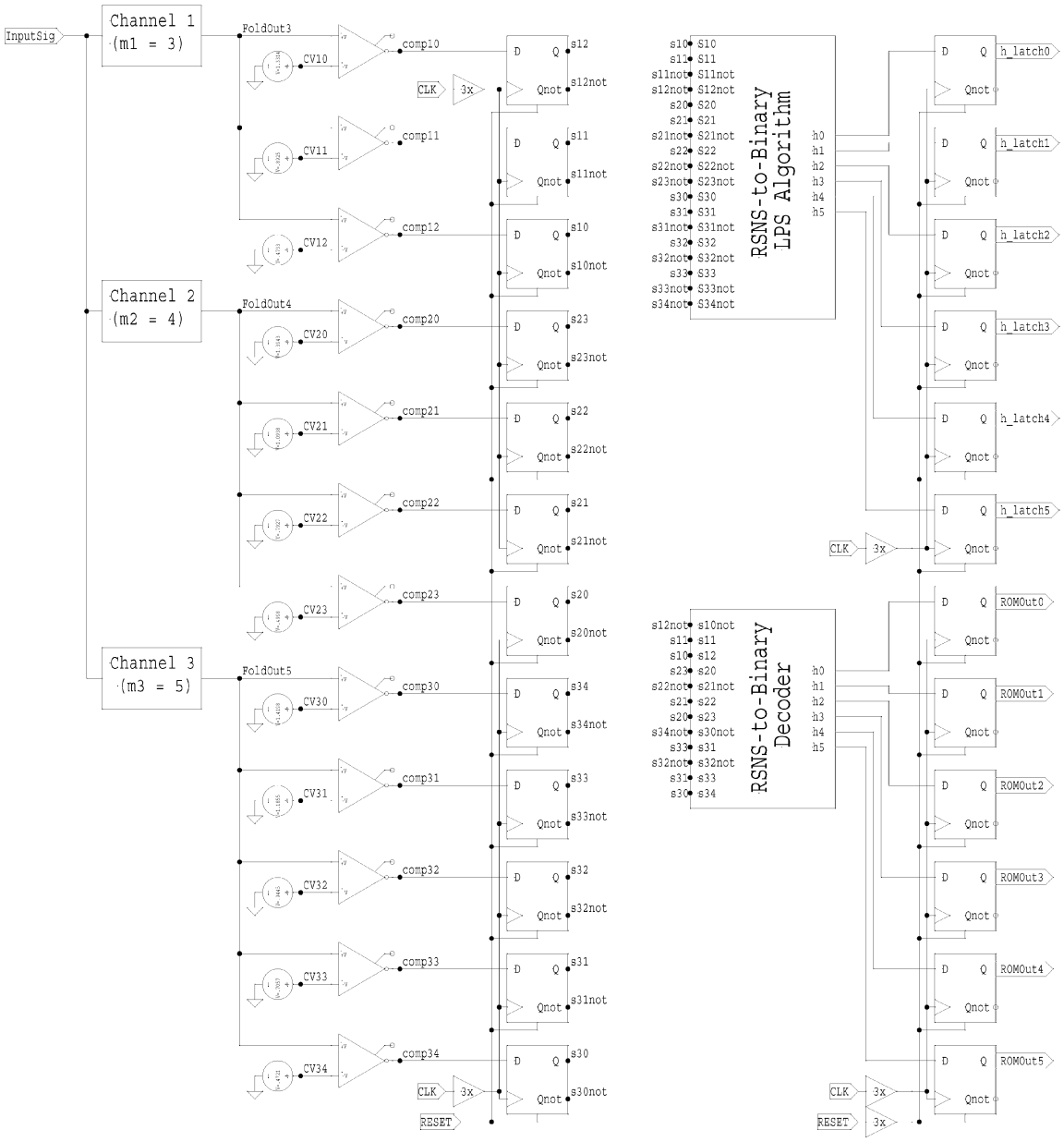

Figure 5.56 Three-channel ADC with decoder converter and LPS converter circuits.

The simulation results for both converters are shown in Figure 5.57. The two conversion methods produced the exact same binary output within the ADC dynamic range. The DNL and INL are the same as shown in Chapter III and are less than one half of an LSB for both conversion methods, as expected.

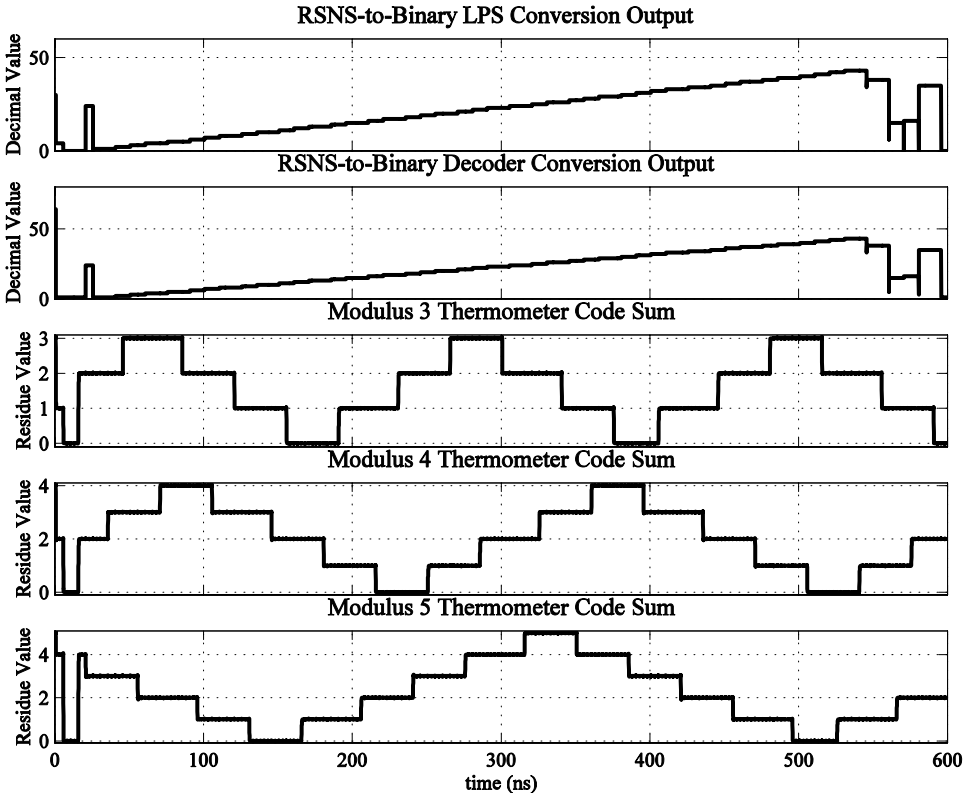

Figure 5.57 Three-channel ADC simulation results for LPS and decoder RSNS-to binary conversion methods.

2. Four-Modulus RSNS-to-Binary Conversion

The RSNS thermometer code output of the four-channel ADC from Chapter III was used to test the four-modulus LPS RSNS-to-binary conversion circuit. The circuit was clocked at 250 MHz because the four-modulus converter design was not pipelined to match the 1-GHz clock used for testing the front end of the four-channel ADC. The simulation results for the four-channel ADC with LPS RSNS-to-binary conversion are shown in Figure 5.58. The binary output is linear, monotonic, and no codes are missing. The DNL and INL are the same as shown in Chapter III.

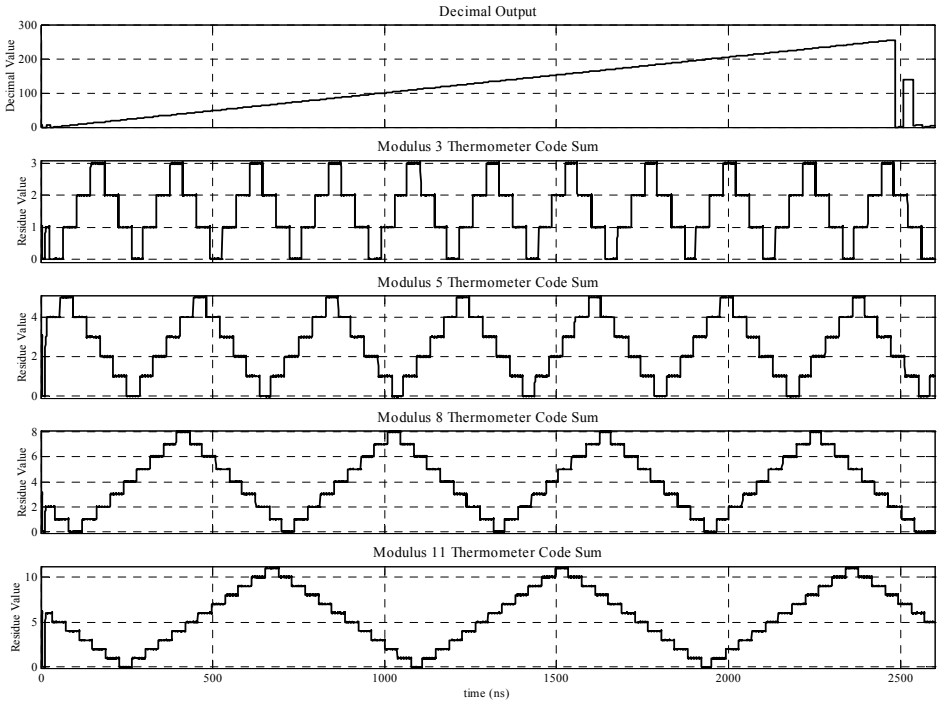

Figure 5.58 Four-channel ADC with RSNS-to-binary conversion.

The four-modulus RSNS-to-binary conversion circuit was also simulated in the same fabrication process as the DIS circuit to see if the conversion circuit would be able to produce *I* and *Q* signals at the rate required by the DIS circuit. The fabrication process used was a Taiwan Semiconductor 0.18-micrometer CMOS process. Since the process was not a mixed-signal process, the four-channel ADC was unable to be used in the simulation. Instead, the SmartSpice simulator generated the ADC thermometer code bits. The results of the simulation are shown in Figure 5.59. The decimal output ramp is perfectly linear, monotonic, and has no missing codes. Furthermore, since perfect thermometer code bits were generated by the simulator, the DNL and INL were both zero at each output code.

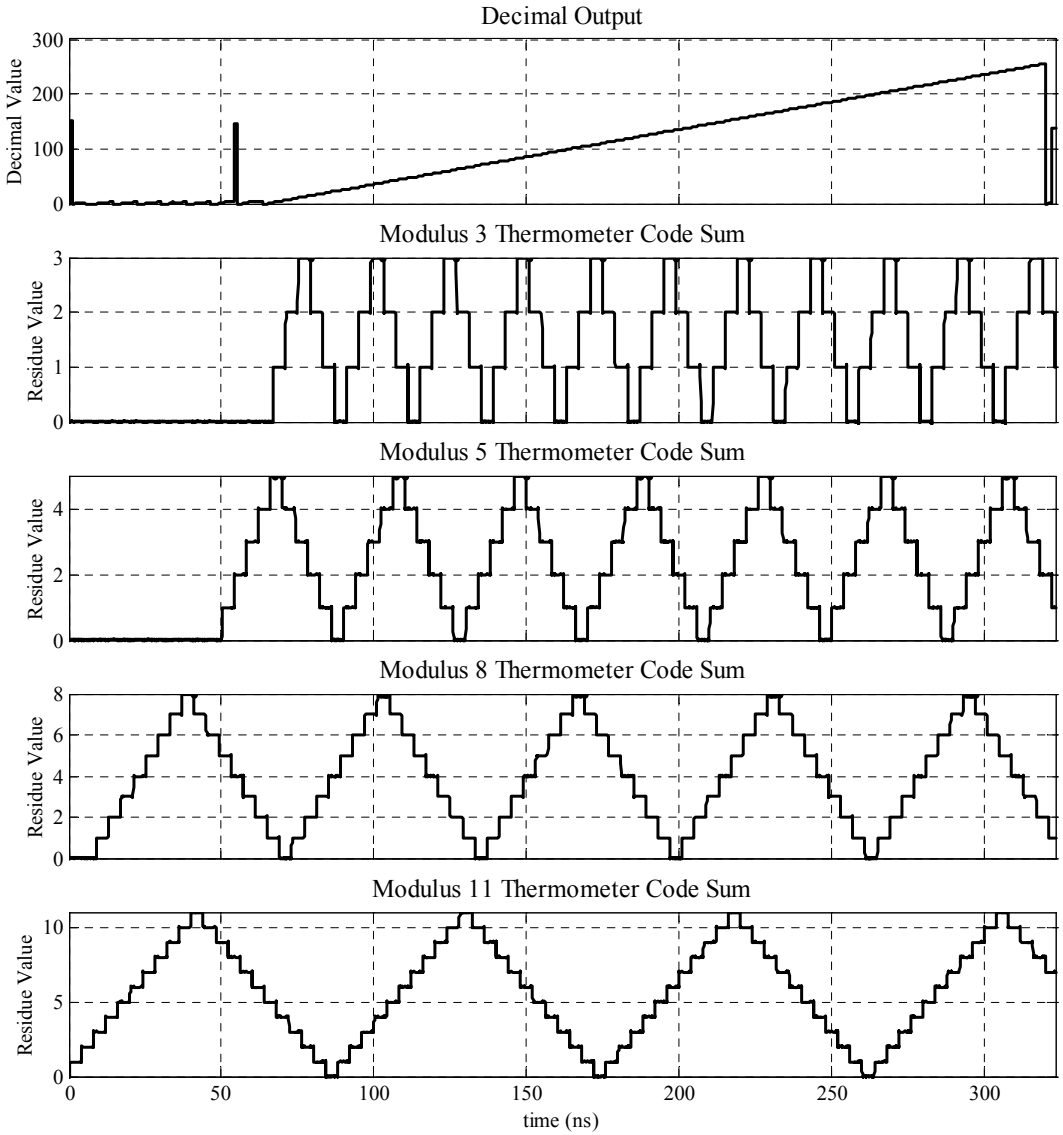

Figure 5.59 Four-modulus RSNS to binary conversion.

G. SUMMARY

This chapter presented three methods to convert RSNS thermometer code to binary. The ROM and decoder conversion methods, although conceptually simple, produced designs that were at least ten to one hundred times larger in terms of transistor count than the LPS conversion method. Furthermore, the LPS conversion method lends itself nicely to pipelining in order to achieve high operation speeds. The circuit schematics were presented for three-modulus and four-modulus RSNS-to-binary converters, and

correct operation was verified by SPICE simulation of the decoders connected to the ADCs from Chapter III. In addition, a simple, detailed design procedure was presented that facilitates the construction of *N*-modulus LPS RSNS-to-binary converters.

The next chapter summarizes the research accomplishments presented in this dissertation. The essence of the original contributions offered in this work reiterated. Suggestions for future work are also provided.

VI. CONCLUSIONS AND FUTURE WORK

This dissertation investigated the concept of an electronic warfare system-on-a-chip that includes a digital image synthesizer capable of producing false targets in response to imaging radar interrogations. In order to make the concept feasible, the DIS was studied in detail to obtain the optimum architecture. A new circuit was then developed for global-clock-signal distribution and synchronization since this problem, if left unresolved, would prevent implementation. Next, a novel compact, high-speed, and low-power RSNS folding ADC design was presented. The ADC is a necessary and critical component of the system-on-a-chip. In the course of the ADC research and design, new three-modulus and *N*-modulus RSNS theory was developed and used to create analytic formulas and an efficient search algorithm to compute the largest sequence of unique RSNS vectors. The remainder of this chapter describes the research and conclusions offered in this dissertation and outlines areas of future work.

A. CONCLUSIONS

Chapter II analyzed the current DIS architecture. Two particular computational redundancies were discovered that, if replaced with the mathematically equivalent circuits provided in the chapter, would result in a DIS design with a significantly reduced layout area and reduced power consumption. In addition, it was shown that the clock distribution scheme in the current DIS was inefficient and not compatible with a SoC design process. A novel automatic clock synchronization and skew control scheme was introduced that decreased SoC design time and increased counter-click pipelined circuit reliability. The C^2 pipeline synchronization theory, circuit schematics, and simulation results provided a comprehensive description and verification of the automatic synchronization design. Furthermore, fabrication and testing of the proof-of-concept automatic clock synchronization chip provided additional verification of the design.

Chapter III presented the circuit design for two RSNS folding ADCs. Both designs were simulated using a SiGe mixed-signal fabrication process used for commercial SoC products. The circuit design for the folding amplifiers, individual folding stages, and latched comparators used in the RSNS folding ADC were described in detail. The three-channel RSNS ADC design procedure was extended to construct a four-channel, eight-bit RSNS folding ADC for converting the analog radar interrogation pulses to the eight-bit digital form required by the DIS. Comprehensive circuit schematics were provided for both ADC designs. Correct operation was verified by simulation results.

Chapter IV extended the previous two-modulus RSNS research into detailed mathematical descriptions for the three-modulus and *N*-modulus RSNS. Using the theory developed in the chapter, a new expression for the $\hat{M}$ length and position for the popular set of moduli $m-1,\ m,\ m+1$ (for *m* even) was presented. A novel GIS-based RSNS circle representation was developed to provide a format more suitable to visual analysis of the RSNS pattern recognition. As a result, enough exploitable properties of the RSNS were discovered that enabled the development of a new and efficient $\hat{M}$ search algorithm, *SmartSearch*. The speed and memory efficiency of *SmartSearch* is several orders of magnitude above the only other known $\hat{M}$ search algorithm.

Chapter V presented three methods to convert RSNS thermometer code to binary. The ROM and decoder conversion methods, although conceptually simple, produced designs that were at least ten to one hundred times larger in terms of transistor count than the LPS conversion method. Furthermore, the LPS conversion design is easily pipelined to achieve high operation speeds. The circuit schematics were presented for three-modulus and four-modulus RSNS-to-binary converters, and correct operation was verified by SPICE simulation of the decoders themselves and also with the decoders connected to the ADCs from Chapter III. In addition, a general *N*-modulus LPS RSNS-to-binary converter design procedure was presented.

B. FUTURE WORK

There is extensive opportunity for future research in the areas covered by this dissertation. The most obvious is completing the design, fabrication, and testing of the DIS EW SoC. The remaining DAC and high-speed memory components must be designed, or purchased from commercial intellectual property (IP) core manufacturers, and integrated with the DIS, eight-bit RSNS folding ADC, and eight-bit RSNS-to-binary converter.

The automatic clock synchronization scheme presented in Chapter II should be integrated into the current DIS design. The primary modification to the design is that the variable delay module in the clock synchronization circuit is replaced by the delay adjustment elements in each DIS range bin.

Although the mathematical description of the RSNS redundancies in Chapter IV is complete, there is still the task of eliminating the $\hat{M}$ search program and discovering a closed-form analytic solution for the longest sequence of unique RSNS vectors for the general N-modulus case. Furthermore, an overall theorem that encompasses all SNS formulations as special cases is also of interest.

Future work for the ADC designs in Chapter III include design of a more efficient and compact comparator and the layout, fabrication, and testing of a hardware implementation of an RSNS folding ADC. Similarly, the RSNS-to-binary converter designs in Chapter V should be implemented in a field programmable gate array (FPGA) to provide hardware verification of the conversion algorithm.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A CLOCK SYNCHRONIZATION CHIP LAYOUT

The physical layout for the clock synchronization chip is provided in this Appendix. The layout was performed using Silvaco Expert version 3.4.10.R. The first section provides illustrations of the layout for basic circuit elements such as simple logic gates and pads. The second section provides images of the physical layout for intermediate modules such as flip-flops. The final section provides images of all major modules and the pad ring. A legend for the fabrication process layers used in the layout illustrations in this Appendix is provided in Figure A.1.

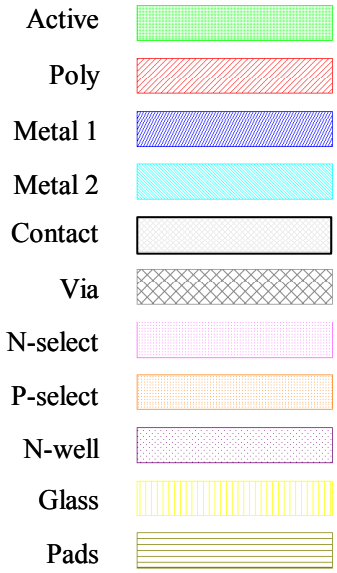

Figure A.1 Fabrication process physical layer legend.

A. BASIC ELEMENT LAYOUT

Layout for the basic circuit elements used in the clock synchronization circuit is shown in Figure A.2. They include a minimum-sized inverter, a double-sized inverter, a passgate, NAND and NOR gates with two, three, four, and five inputs, a buffer, and a two-input XNOR. The figure also shows the layout for a 2-to-1 multiplexer and the tuned four-input NAND (NAND4x2) used in the phase check module.

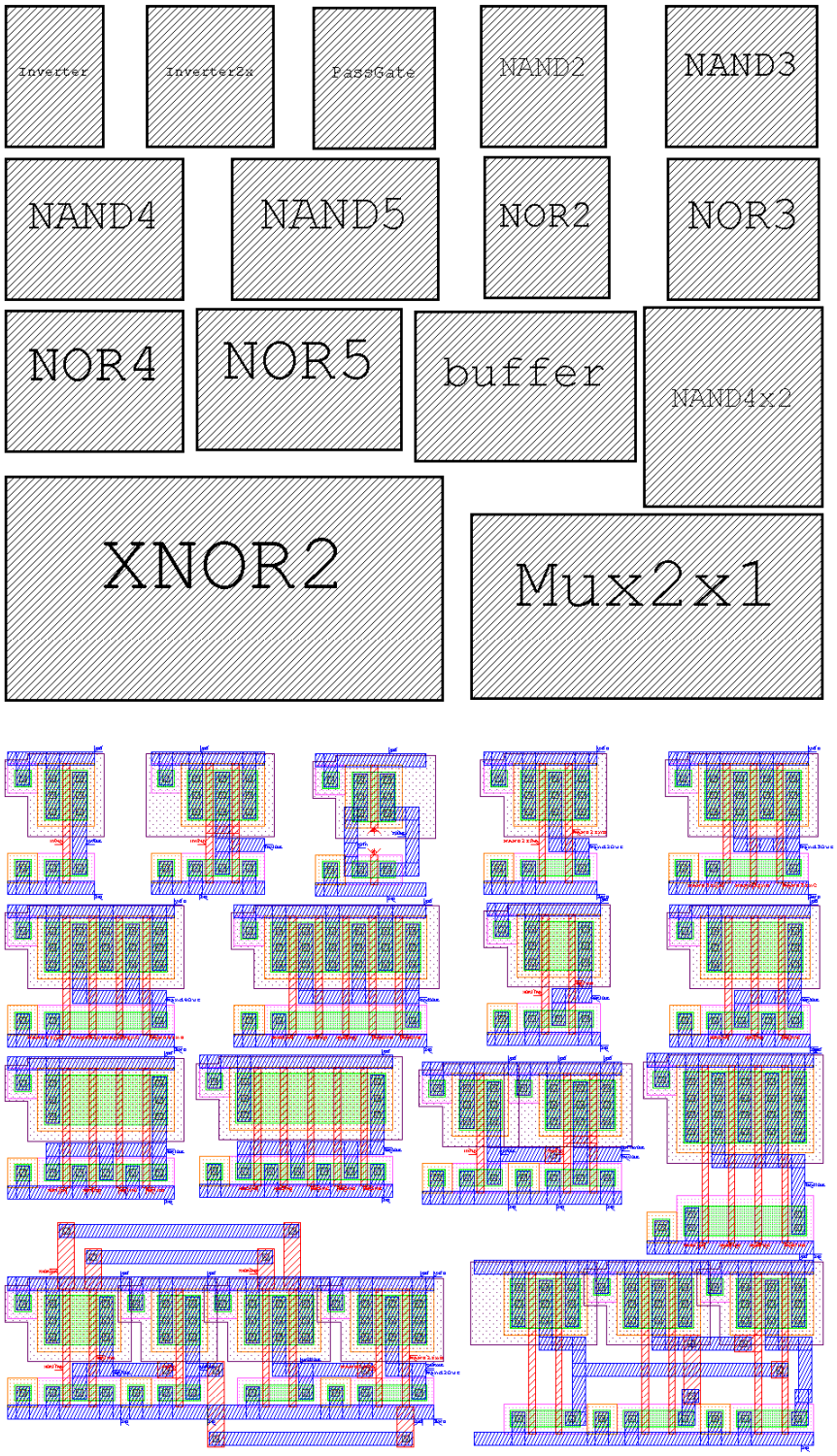

Figure A.2 Physical layout of basic circuit elements.

The top of the figure shows the relative size of the elements as well as their display name. The bottom of the figure shows the detailed layout of the logic elements corresponding to the blocks at the top of the diagram. As much as possible, the physical layout distance between the power and ground supply rails was kept constant and the gate inputs and outputs were placed in uniform position from gate to gate so that they could be connected together without unnecessary additional routing.

The circuit elements used in the pad ring are shown in Figure A.3. They include a driver for driving off-chip signals, a pad for external connections, a receiver for driving on-chip signals, and an electrostatic discharge (ESD) protection circuit.

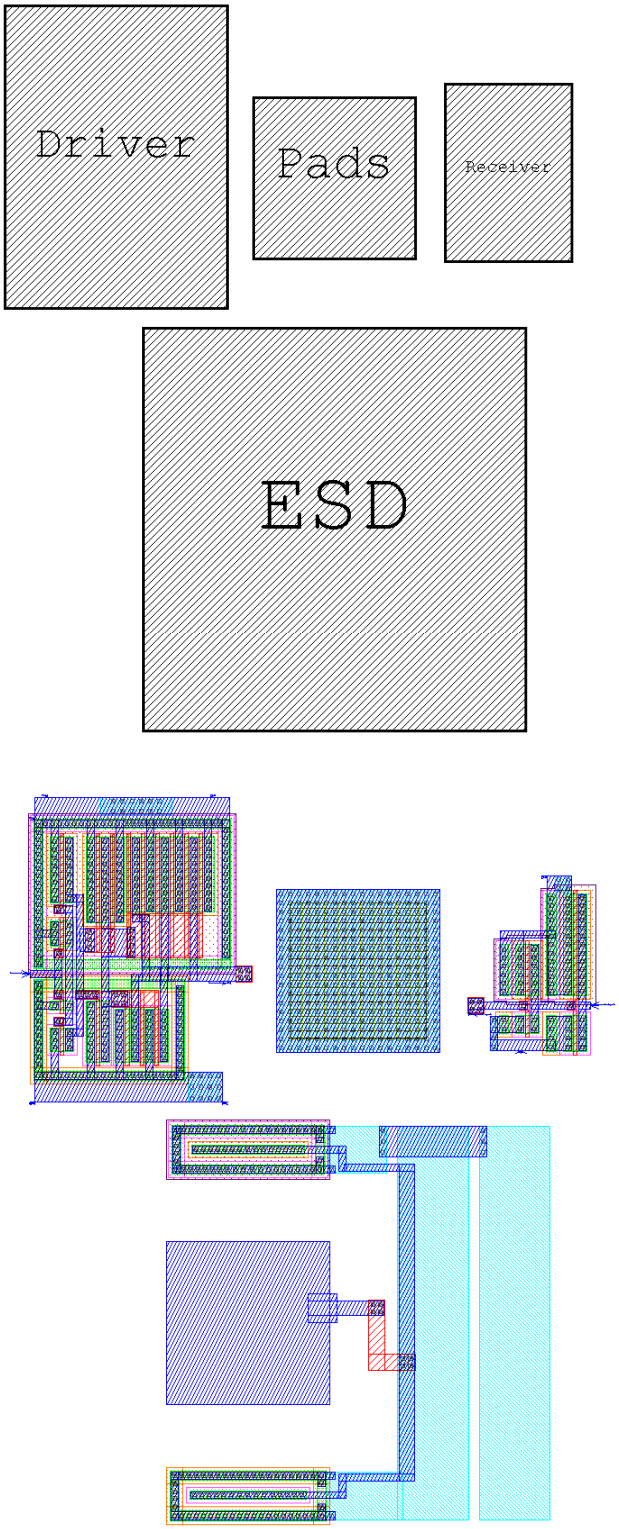

Figure A.3 Physical layout of pad ring elements.

B. INTERMEDIATE COMPONENT LAYOUT

The layout for the intermediate circuit components used in the clock synchronization circuit is provided in this section. The D-flip-flop is shown in block form in Figure A.4 and with details in Figure A.5. Notice the heights of the inverters are the same and the inputs to the inverters are on the left while the outputs are on the right. This enables a compact layout for larger modules with a minimum of additional wiring. Furthermore, modules such as the D-flip-flop can be neatly stacked or attached end-to-end to form larger compact layouts.

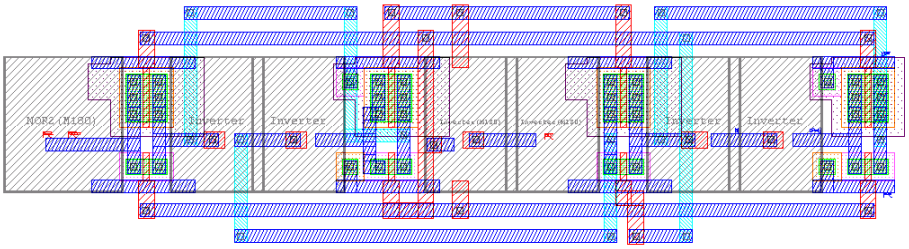

Figure A.4 Block outline of D-flip-flop.

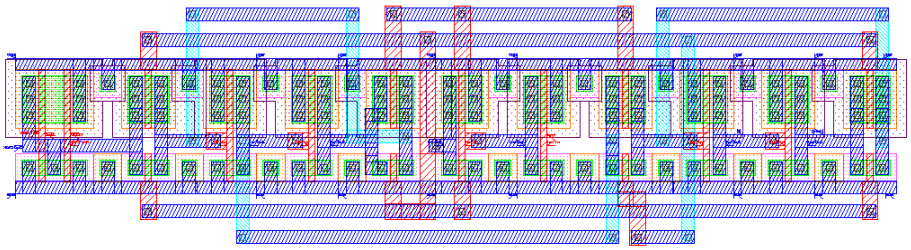

Figure A.5 Detailed layout of D-flip-flop.

The toggle flip-flop is shown in block form in Figure A.6 and with details in Figure A.7. The left-most section of the layout shows the custom-designed function select passgates that form the logic behind the hold, clear, load, and toggle functions of the toggle flip-flop.

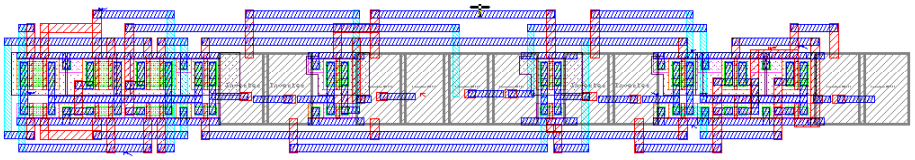

Figure A.6 Block outline of T-type flip-flop.

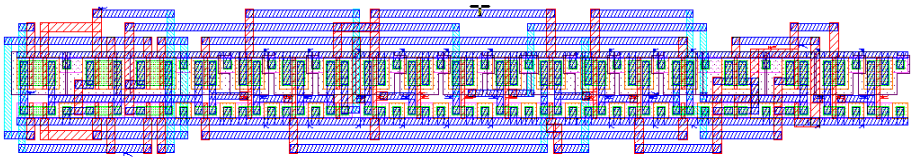

Figure A.7 Detailed layout of T-type flip-flop.

Figure A.8 and Figure A.9 provide the block structure and detailed layout of the 16-to-1 multiplexer delay modules. The inverter chain surrounding the 16-to-1 multiplexer is connected such that there are two minimum-sized inverters between each multiplexer input.

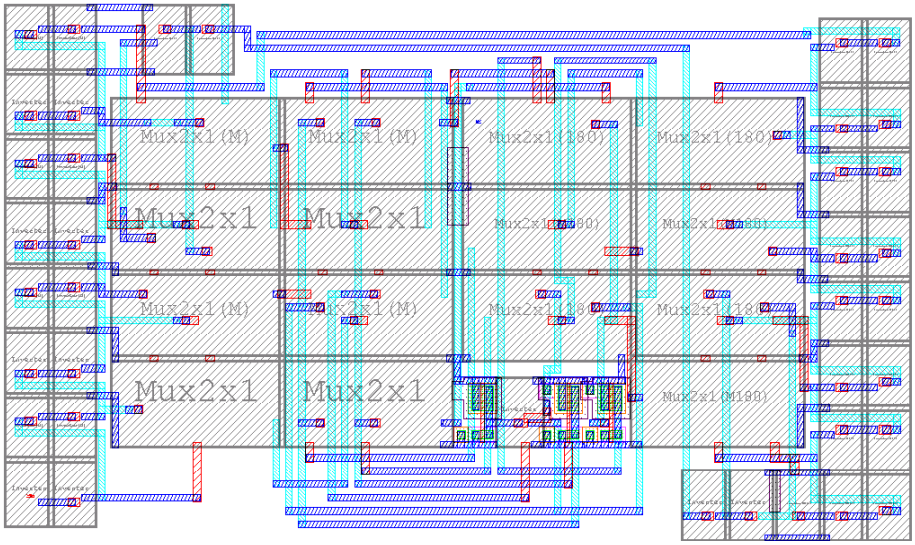

Figure A.8 Block Outline of 16x1 multiplexer delay module.

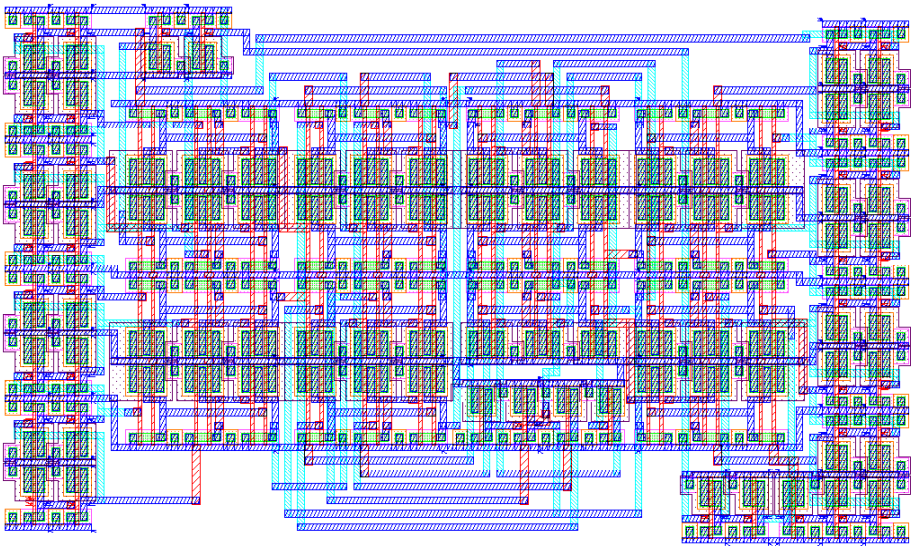

Figure A.9 Detailed layout of 16x1 multiplexer.

Figure A.10 and Figure A.11 provide the block and detailed layout of the five-bit counter, which is primarily composed of four toggle flip-flops and a few two-input NAND and NOR gates.

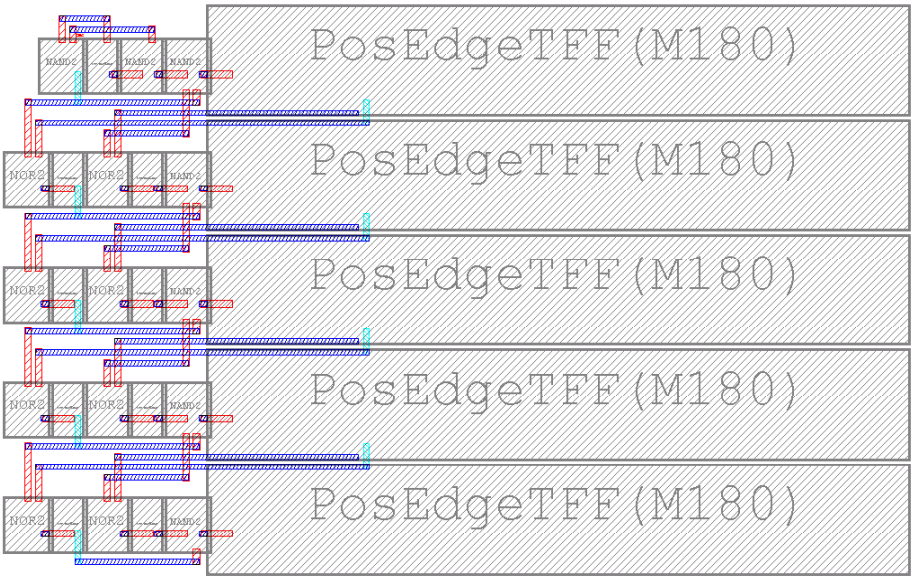

Figure A.10 Block outline of five-bit counter.

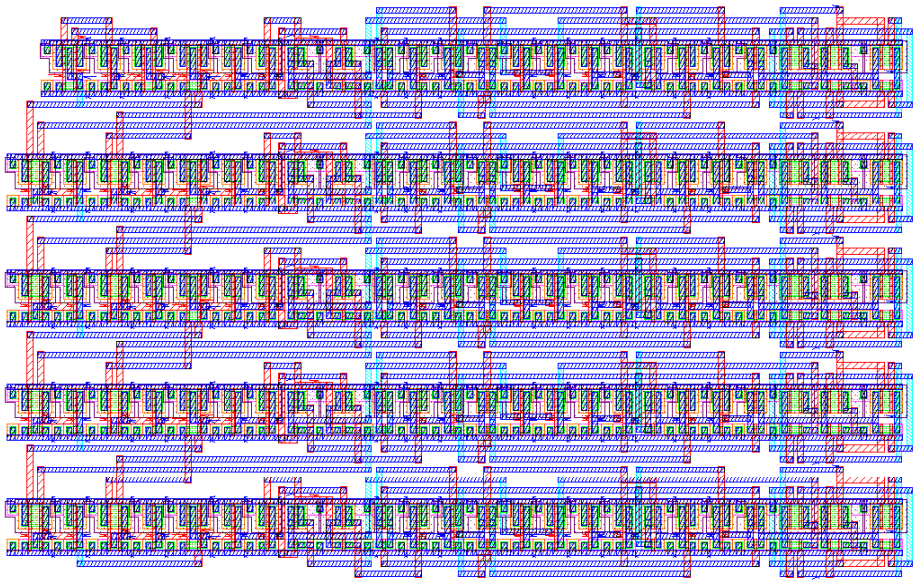

Figure A.11 Detailed layout of five-bit counter.

Finally, Figure A.12 provides the layout of the pad ring that surrounds the clock synchronization circuit. Only 26 of the 40 pins were used for input and output signals, power, and ground. The nine driver circuits for driving signals off-chip are clearly visible on the diagram. The four corner pads supply power and ground to the pad ring elements.

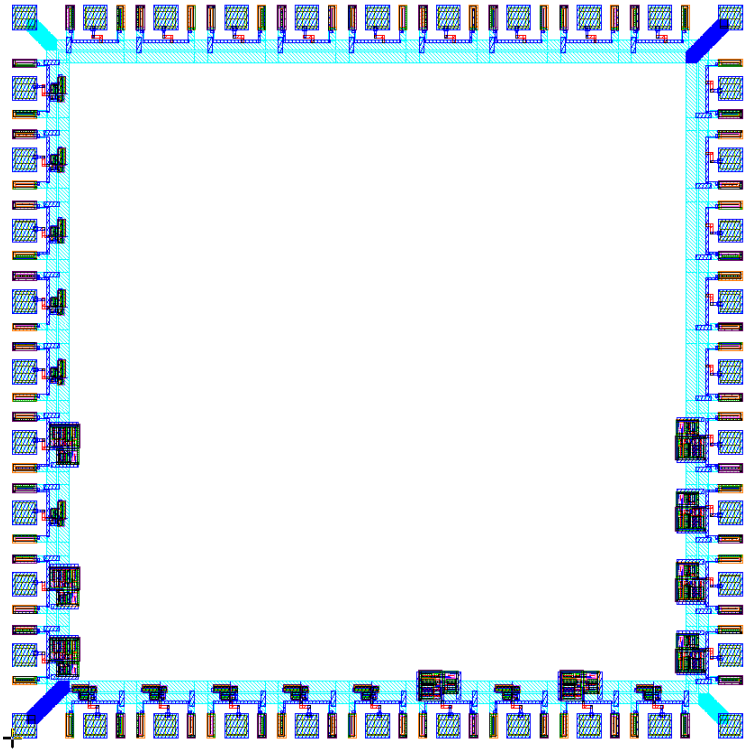

Figure A.12 Detailed layout of the synchronization chip pad ring.

C. MAJOR COMPONENT LAYOUT

The layout for the major components used in the clock synchronization circuit is provided in this section. They include the FSM, phase-check circuit, five-bit wraparound counter, and variable delay module. Finally, the layout for the entire chip is presented at the end of the section along with a photograph of the fabricated chip. The FSM is shown in block form in Figure A.13 and with details in Figure A.14.

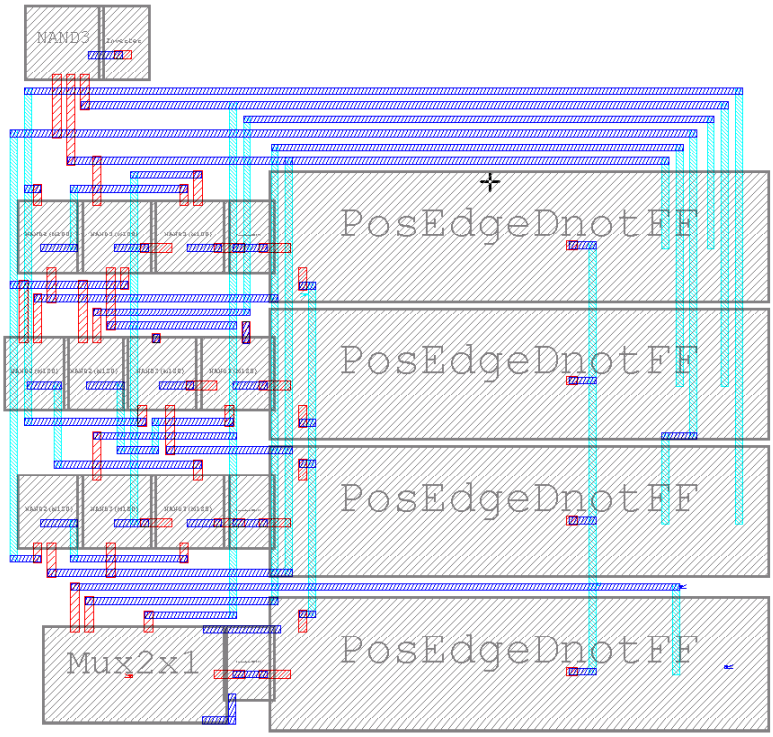

Figure A.13 Block outline of finite state machine.

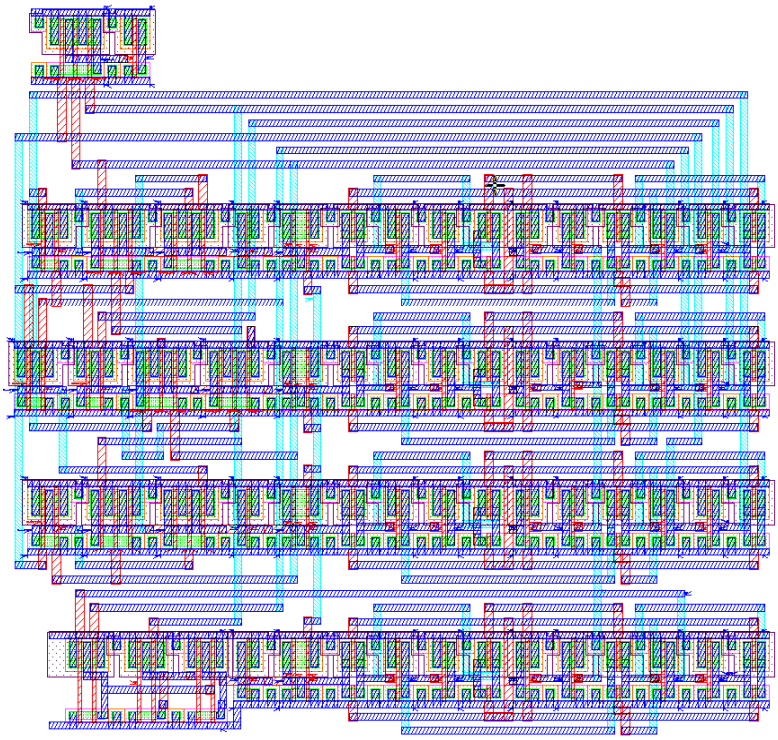

Figure A.14 Detailed layout of finite state machine.

Figure A.15 and Figure A.16 provide the block and detailed layout of the phase-check circuit. In the center of the circuit is the tunable NAND4x2 that is the heart of the phase-check circuit.

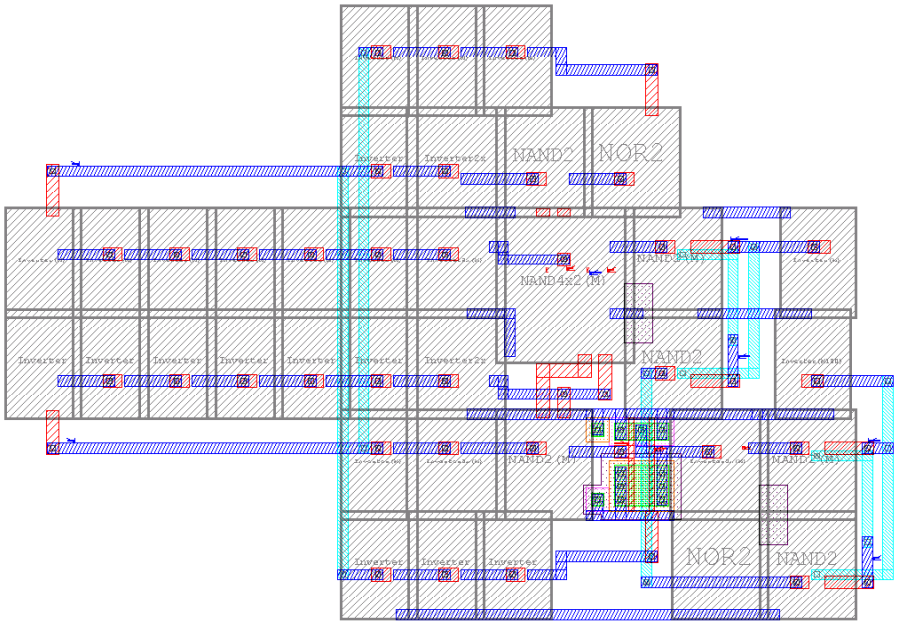

Figure A.15 Block outline of phase-check circuit.

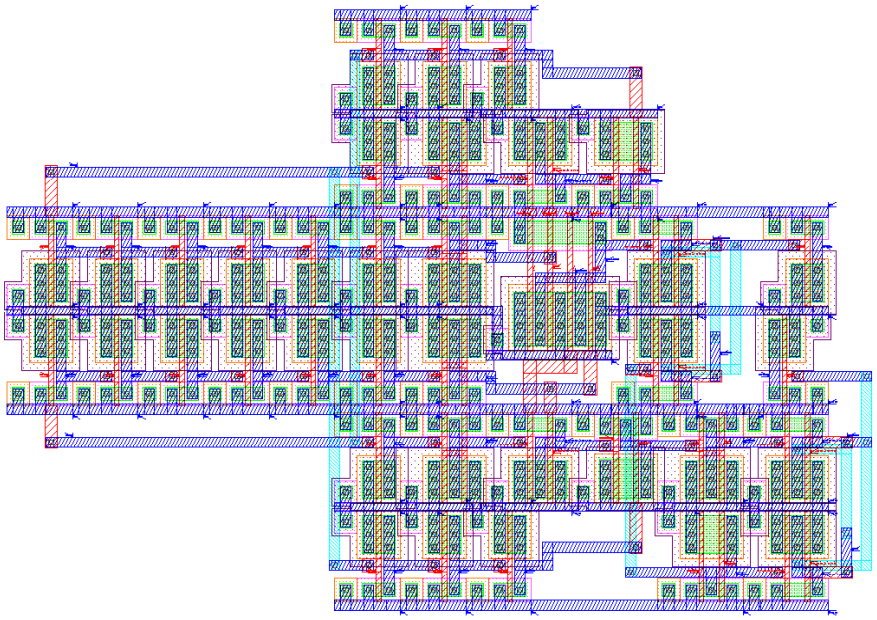

Figure A.16 Detailed layout of phase-check circuit.

Figure A.17 and Figure A.18 provide the block and detailed layout of the five-bit wraparound counter. The circuit primarily consists of the five-bit counter from the previous section with additional logic to implement the roll over function.

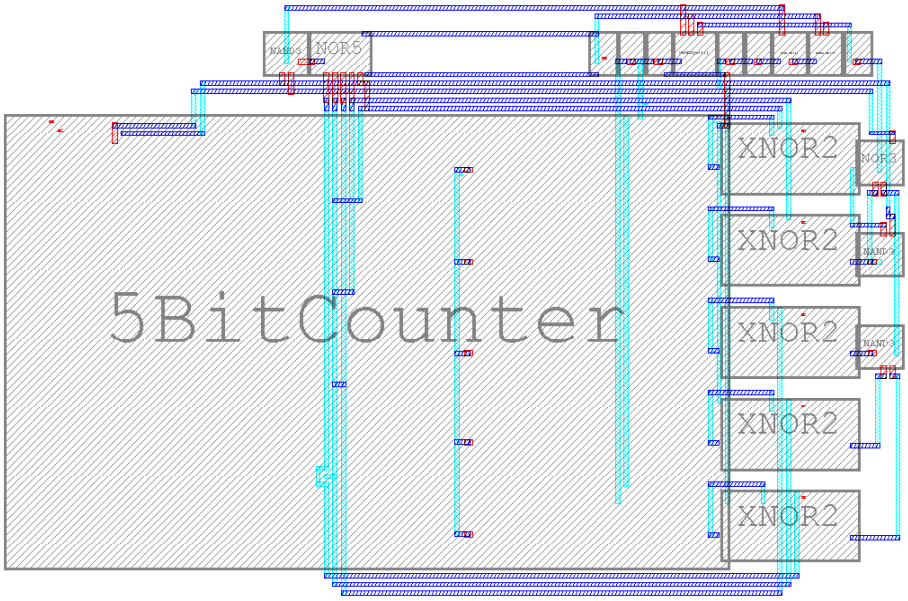

Figure A.17 Block outline of five-bit counter with wraparound.

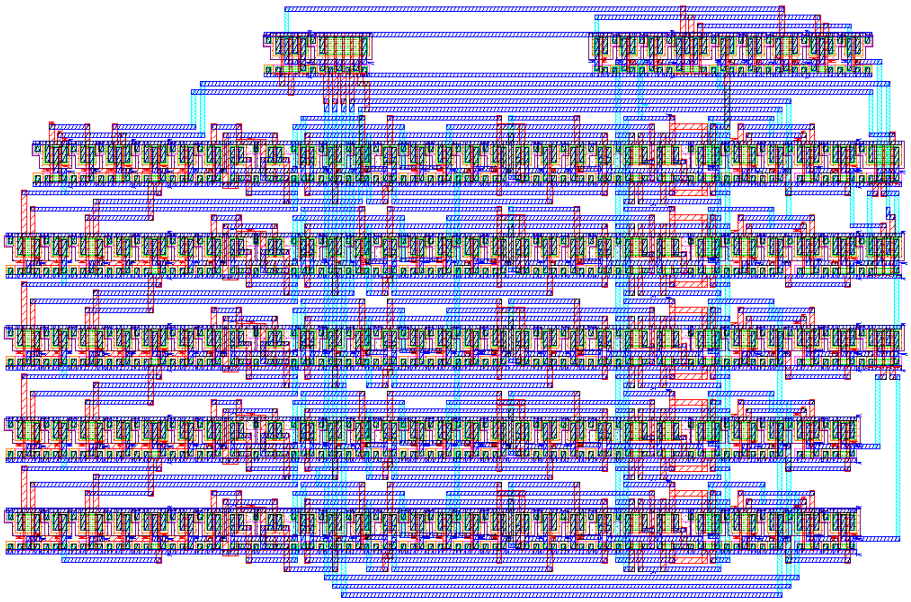

Figure A.18 Detailed layout of five-bit counter with wraparound.

Figure A.19 and Figure A.20 provide the block and detailed layout of the variable delay module. The circuit consists two 16-to-1 multiplexer delay modules connected in series. No additional glue logic is required.

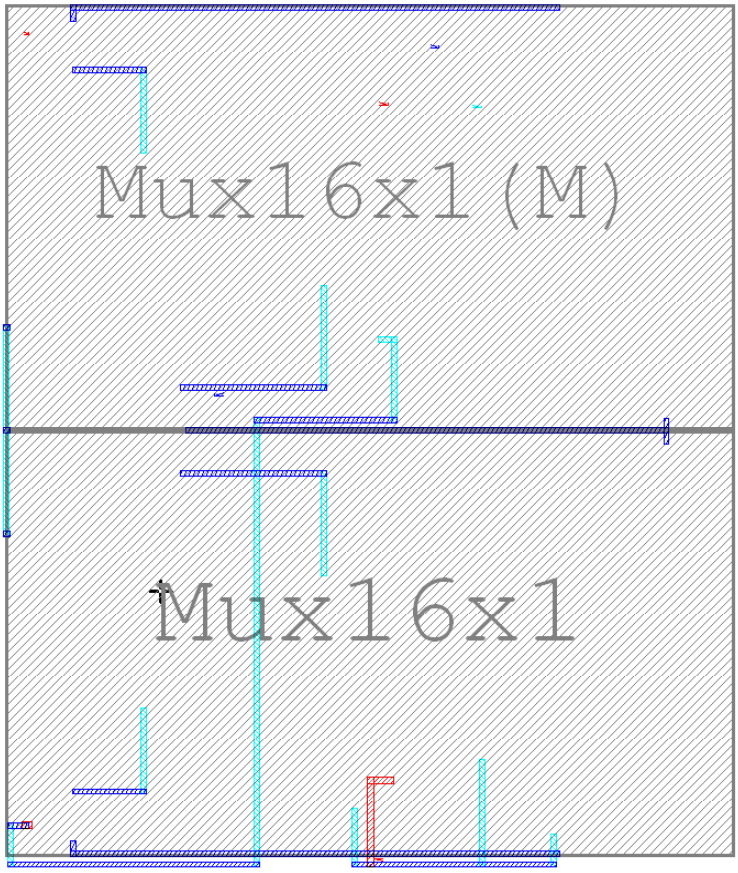

Figure A.19 Block outline of variable delay module.

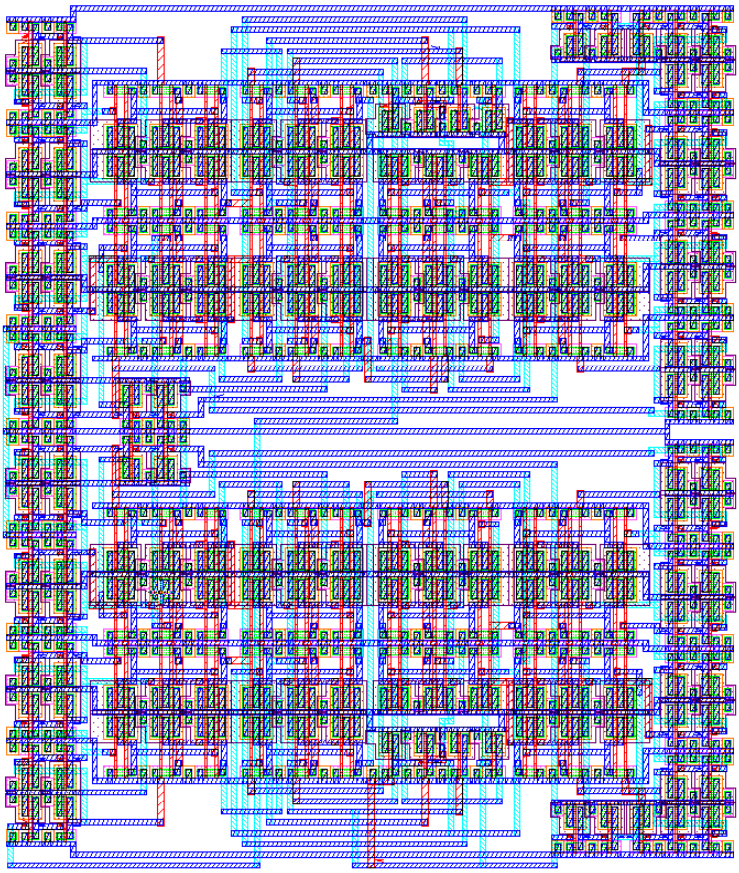

Figure A.20 Detailed layout of variable delay module.

Figure A.21 provides detailed physical layout of the entire proof-of-concept clock synchronization test chip. The two variable delay modules are on the left of the diagram, the counter is in the center, and the FSM and phase-check module are on the right. Notice that the design uses less than half of the 2.2 by 2.2 square millimeter chip cavity.

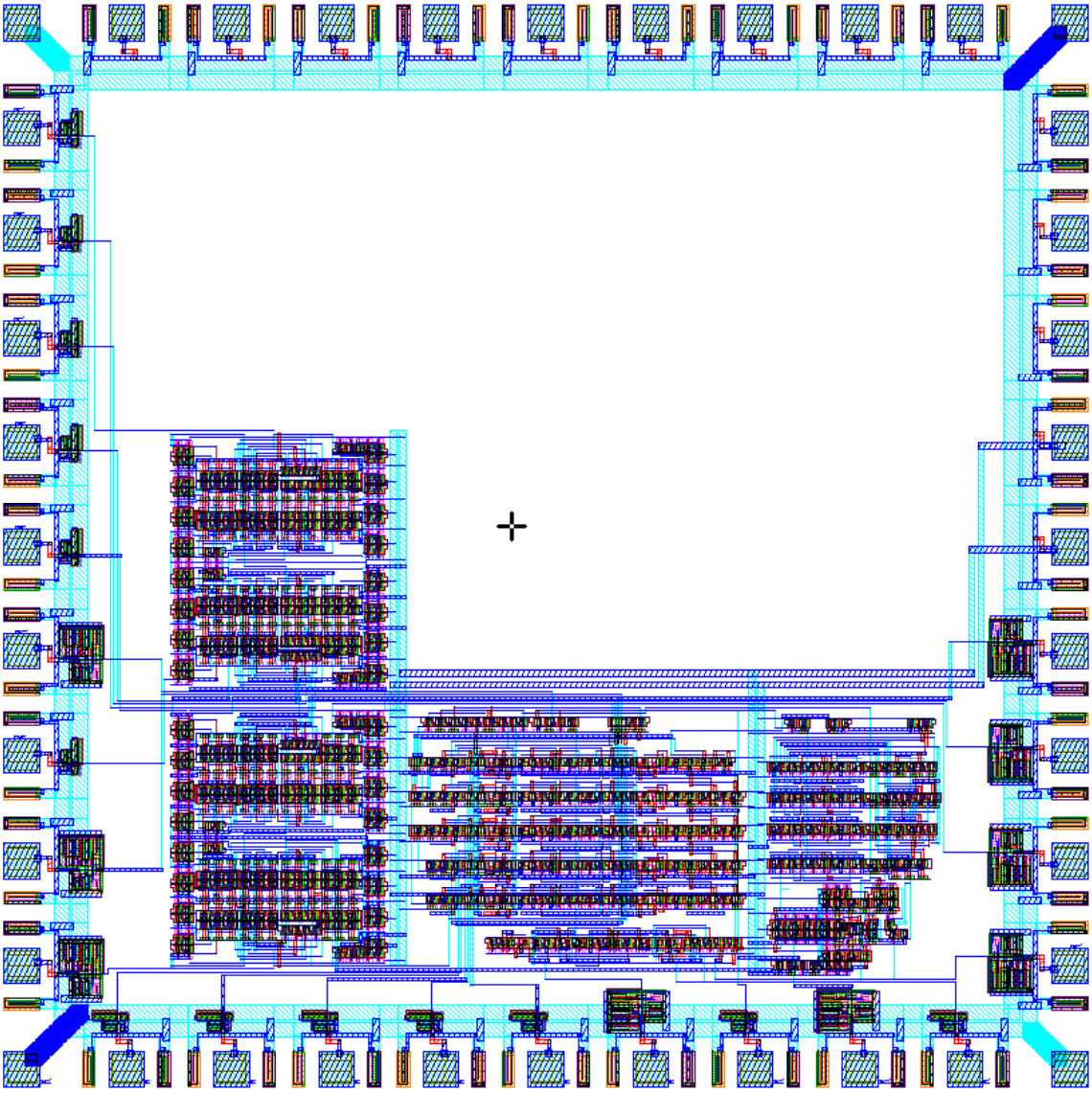

Figure A.21 Detailed layout of entire chip and pad ring.

Figure A.22 is a picture of the fabricated chip and bond wires in the ceramic packaging. Figure A.23 is the same picture with the center enlarged approximately six times so that the structure of the layout from Figure A.21 is visible.

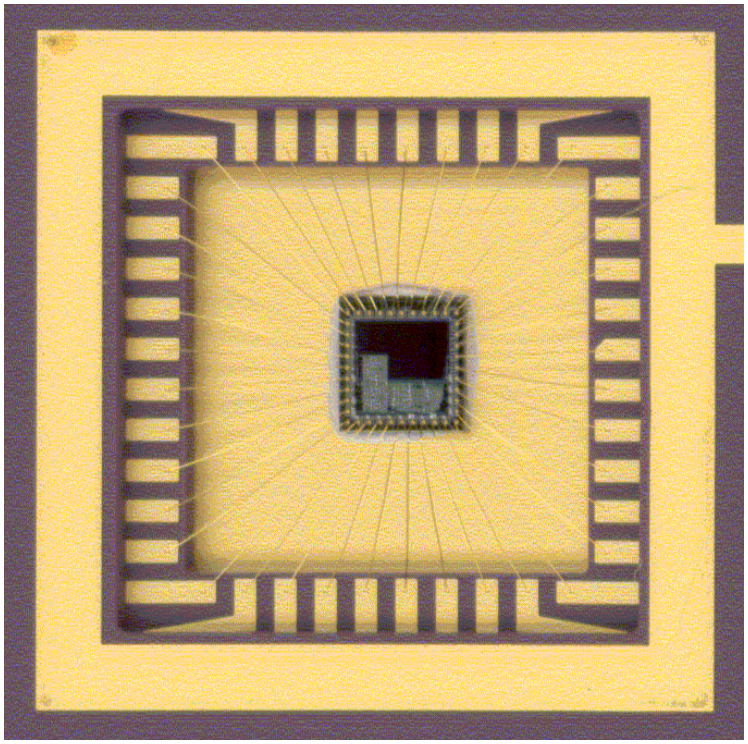

Figure A.22 Chip photograph.

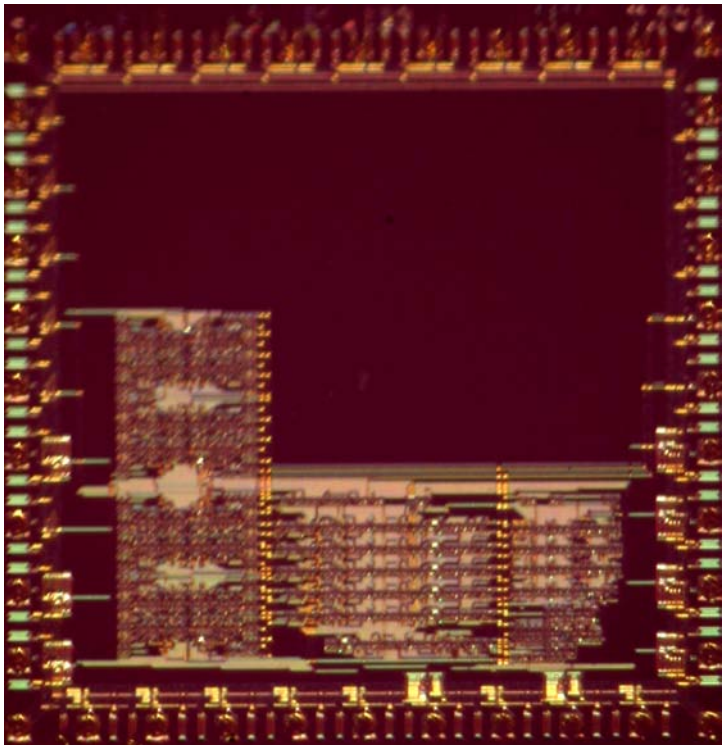

Figure A.23 Enlarged chip photograph showing circuit layout and pad ring detail.

APPENDIX B CLOCK SYNCHROMIZATION DETAILED SIMULATION RESULTS

The first section in this Appendix provides detailed simulation data on all of the logic gates employed in the design of the clock synchronization chip. The second section provides schematics and verification simulation results for the intermediate circuit elements used compose the major modules described in Chapter II. All simulations were performed using both Silvaco Parallel SmartSpice version 2.6.0.R and Tanner T-Spice Pro version 7. The simulation results for each gate or module were consistent for both versions of SPICE. The simulation results provided in this section are from the Silvaco software. The device models and model parameters were provided by MOSIS for the AMI Semiconductor (AMIS) ABN process [26]. The model parameters obtained from MOSIS were the average of measured parameters taken from test structures on several wafers in the same lot as the wafer containing the clock synchronization chip.

A. BASIC LOGIC GATES

The logic gates from the clock synchronization design are listed in the table in Figure B.1.

Logic gate	NMH (V)	NML (V)	RISE (s)	FALL (s)	TPLH (s)	TPHL (s)	PWR (W)
Inverter	1.73	1.18	7.11E-10	9.28E-10	3.96E-10	5.33E-10	3.72E-03
Inverter2x	1.72	1.16	4.87E-10	6.21E-10	2.85E-10	4.17E-10	5.98E-03
Buffer	2.72	2.35	4.92E-10	5.47E-10	7.00E-10	6.64E-10	5.93E-03
NAND2	1.75	0.88	7.49E-10	1.66E-09	4.20E-10	8.08E-10	3.51E-03
NAND3	1.98	0.55	7.91E-10	2.47E-09	4.39E-10	1.15E-09	3.33E-03
NAND4x2	1.56	0.67	7.08E-10	1.67E-09	4.16E-10	8.22E-10	4.03E-03
NOR2	1.52	1.18	1.39E-09	1.09E-09	7.53E-10	6.35E-10	2.75E-03
NOR3	1.32	1.22	2.19E-09	1.28E-09	1.19E-09	7.07E-10	2.74E-03
NOR5 (4)	1.8	0.57		1.57E-09	2.20E-09	7.64E-10	2.64E-03
NOR5 (2)	1.8	0.57	2.64E-09	1.12E-09	1.50E-09	5.46E-10	2.64E-03
XNOR2	2.6	2.35	7.37E-10	1.66E-09	1.14E-09	1.48E-09	3.92E-03
Passgate (4)			1.79E-09	1.58E-09	4.60E-10	3.30E-10	2.30E-04
Passgate (2)			1.09E-09	9.31E-10	2.44E-10	1.82E-10	2.30E-04

Figure B.1 Detailed simulation data for clock synchronization chip logic gates.

The columns in the figure are noise margin high (NMH) and noise margin low (NML), rise time (RISE) and fall time (FALL), gate delay propagating low-to-high (TPLH) and high-to-low (TPHL), and the power consumed by the gate (PWR). The correct operation of the logic gates was verified using simulations with 100-MHz, 50%-duty-cycle square-wave input signals. The rise and fall times of the input signals were approximately one nanosecond. For each simulation, the output of each gate was loaded with four minimum-sized inverters except for the passgate and NOR5, which were simulated with four inverter loads as well as two inverter loads. NMH and NML do not apply to the passgate and so are represented by blank cells figure. The blank cell representing the NOR5 rise time with four inverter loads is due to the fact that the logic gate could not achieve a logic high under the stimulation conditions described above. Graphs of the data in Figure B.1 are provided in Figure B.2, Figure B.3, Figure B.4, and Figure B.5.

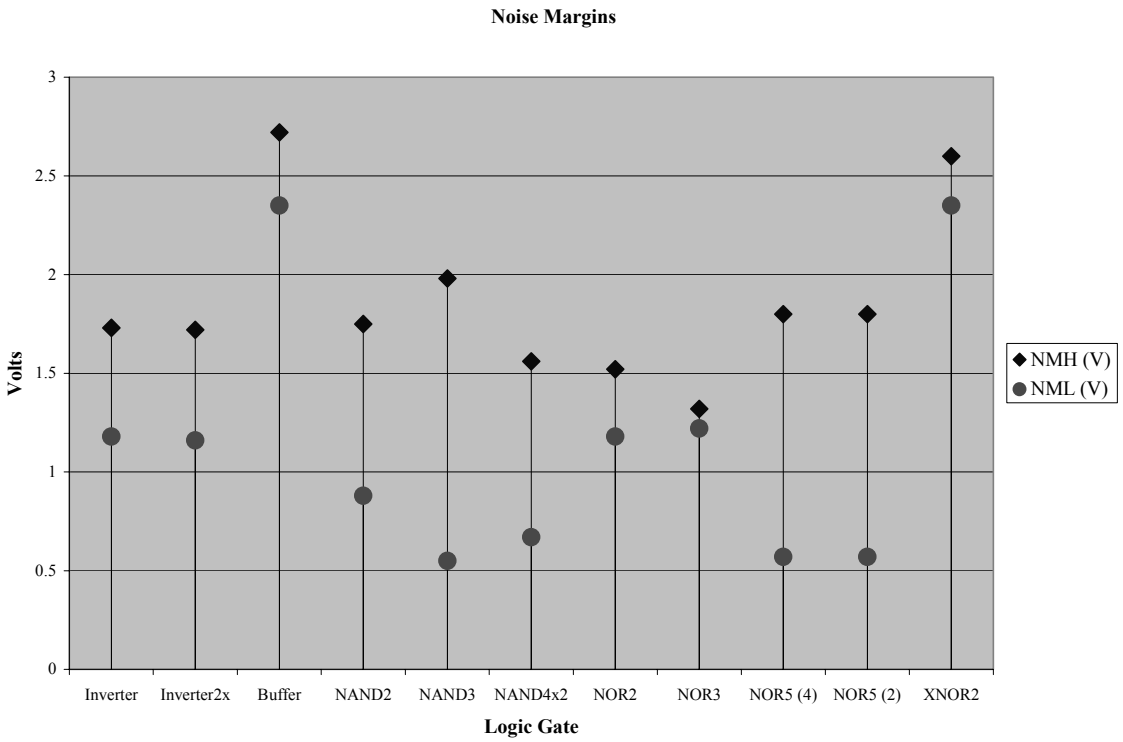

Figure B.2 Logic gate noise margins.

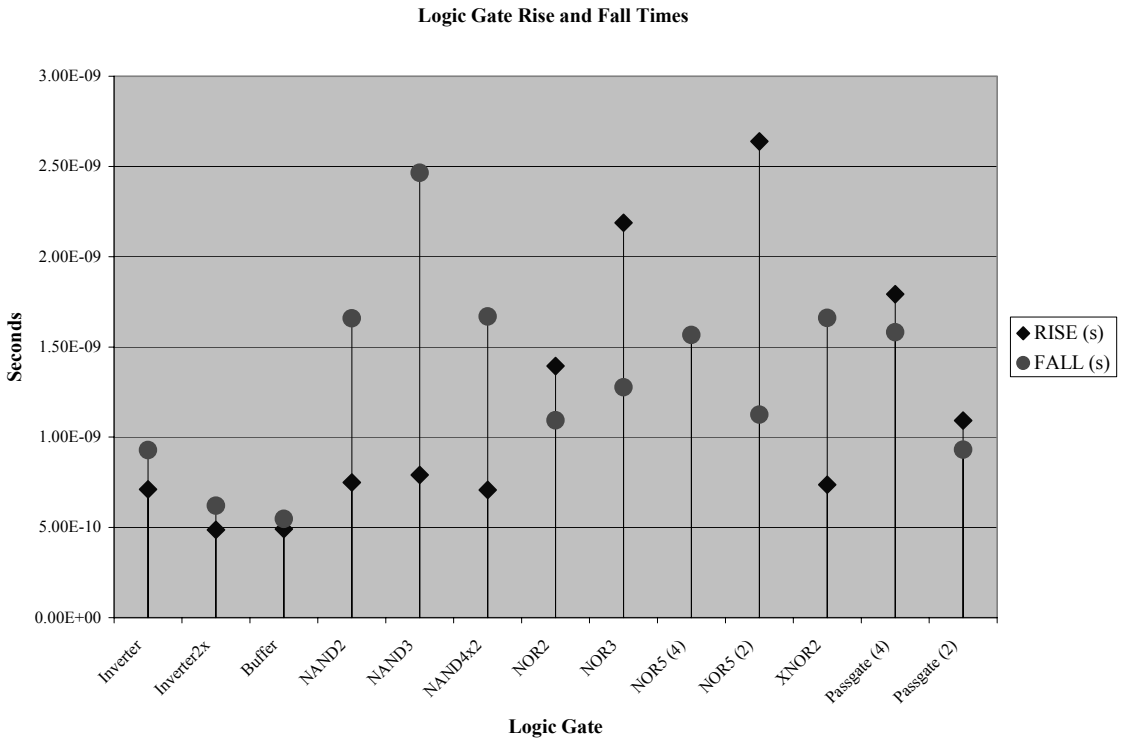

Figure B.3 Logic gate rise and fall times.

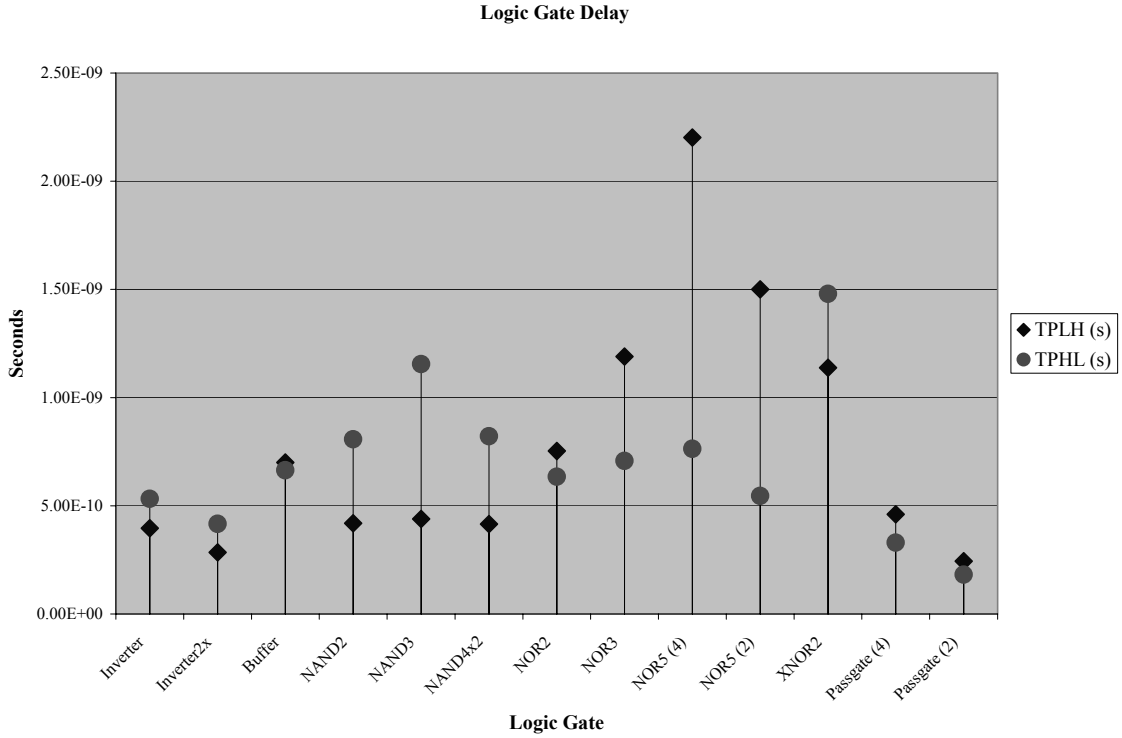

Figure B.4 Logic gate delay.

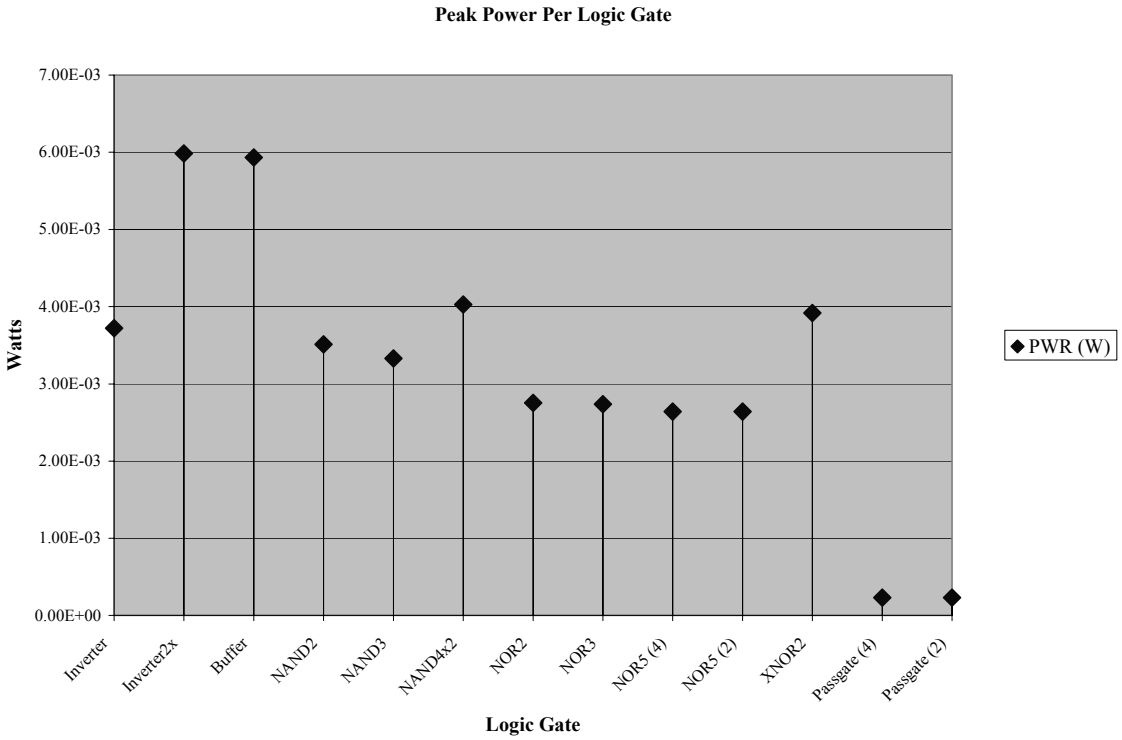

Figure B.5 Graph of peak power for each logic gate.

B. INTERMEDIATE CIRCUIT COMPONENTS

The intermediate circuit components are the circuits used to the major modules of the clock synchronization chip described in Chapter II. They include the 2-to-1 multiplexer, D-flip-flop and toggle flip-flop, and 16-to-1 multiplexer. Complete schematics and verification simulation results are provided in this section for the intermediate circuit components. In addition, power dissipation simulation results are provided at the end of this section for all intermediate circuits as well as for the major modules.

The schematic for the 2-to-1 multiplexer circuit is given in Figure B.6. Although this design is not as compact as passgate-based multiplexers, it provides better drive capability and a more predictable path delay when used as a building block in a larger multiplexer. The operation of the two-to-one multiplexer circuit was verified by simulation and a plot of selected simulation results is provided in Figure B.7.

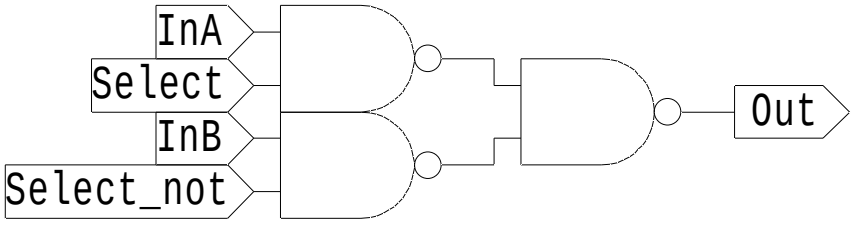

Figure B.6 Schematic of the 2-to-1 multiplexer circuit.

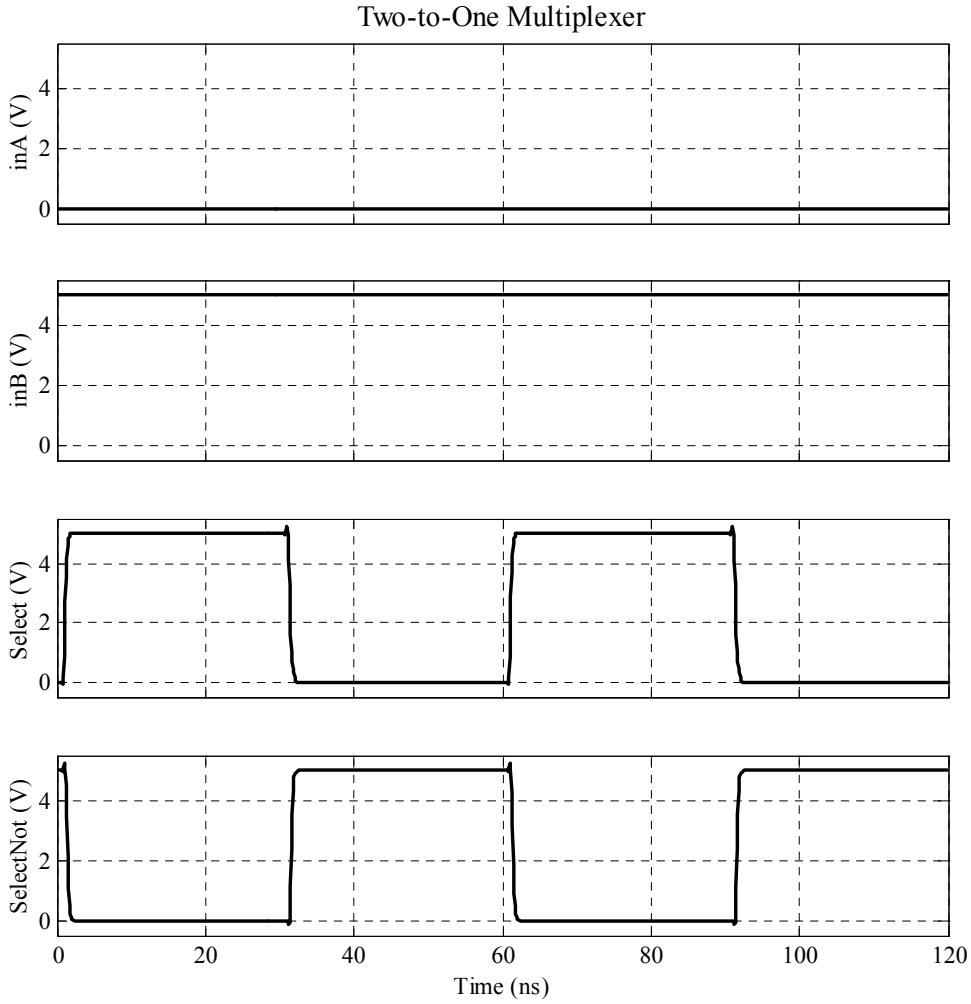

Figure B.7 Simulation results for the 2-to-1 multiplexer circuit.

The schematic of the D-flip-flop is provided in Figure B.8. This is a fairly standard design for this type of flip-flop and includes logic for a synchronous reset capability. Simulation results for this circuit are displayed in Figure B.9.

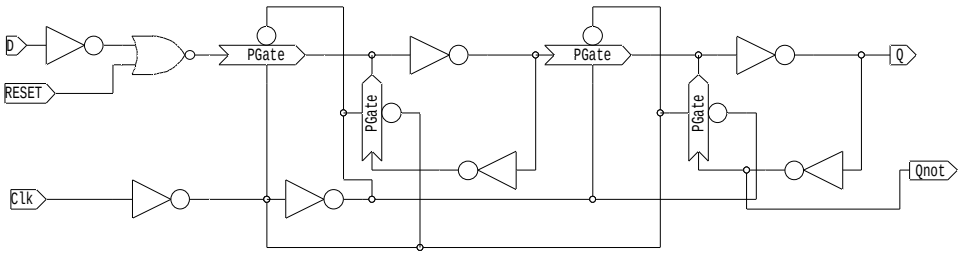

Figure B.8 Schematic of D- flip-flop circuit.

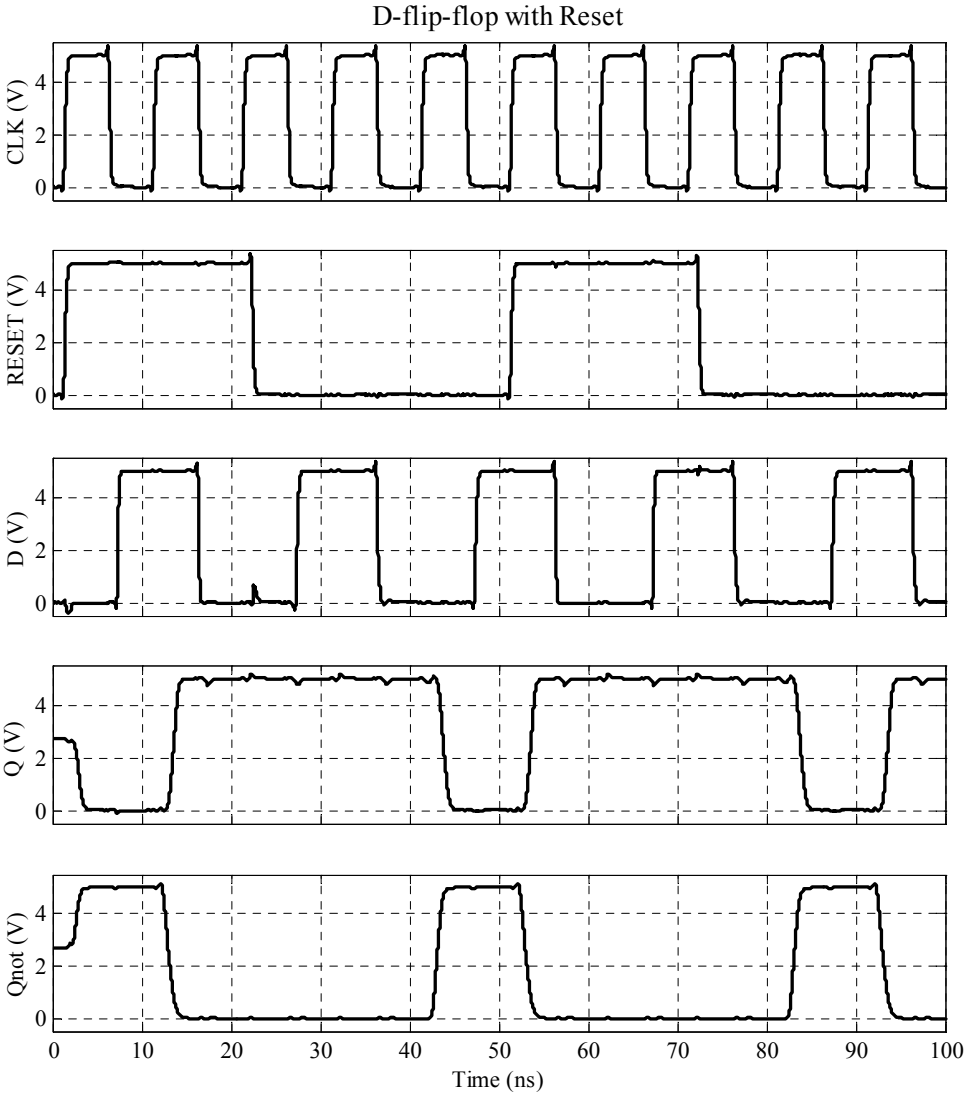

Figure B.9 Simulation results for the D-flip-flop circuit.

The schematic of the toggle flip-flop is provided in Figure B.10. This design is unique in that the input to the flip-flop is selectable. Using passgates with dual select inputs, the flip-flop can perform the functions toggle, hold, clear, and load data. Simulation results for this circuit are given in Figure B.11.

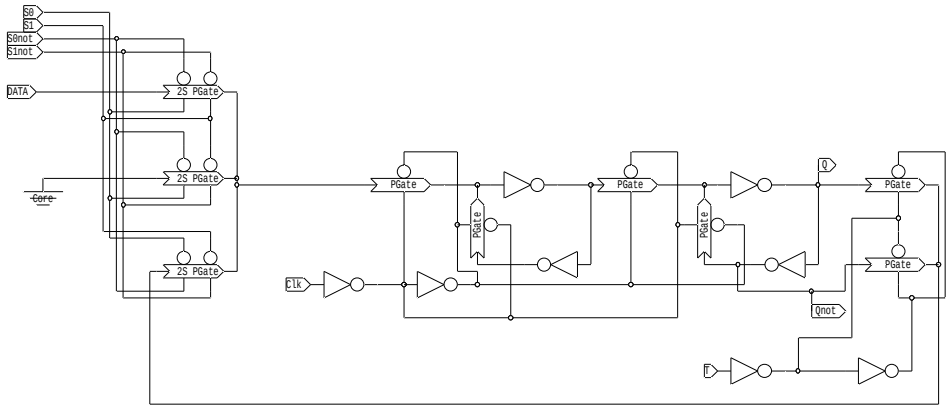

Figure B.10 Schematic of toggle flip-flop circuit.

Figure B.11 Simulation results for the toggle flip-flop circuit.

The schematic for the 16-to-1 multiplexer circuit is given in Figure B.12. This is a fairly simple design that uses 15 2-to-1 multiplexers to form the 16-to-1 multiplexer circuit.

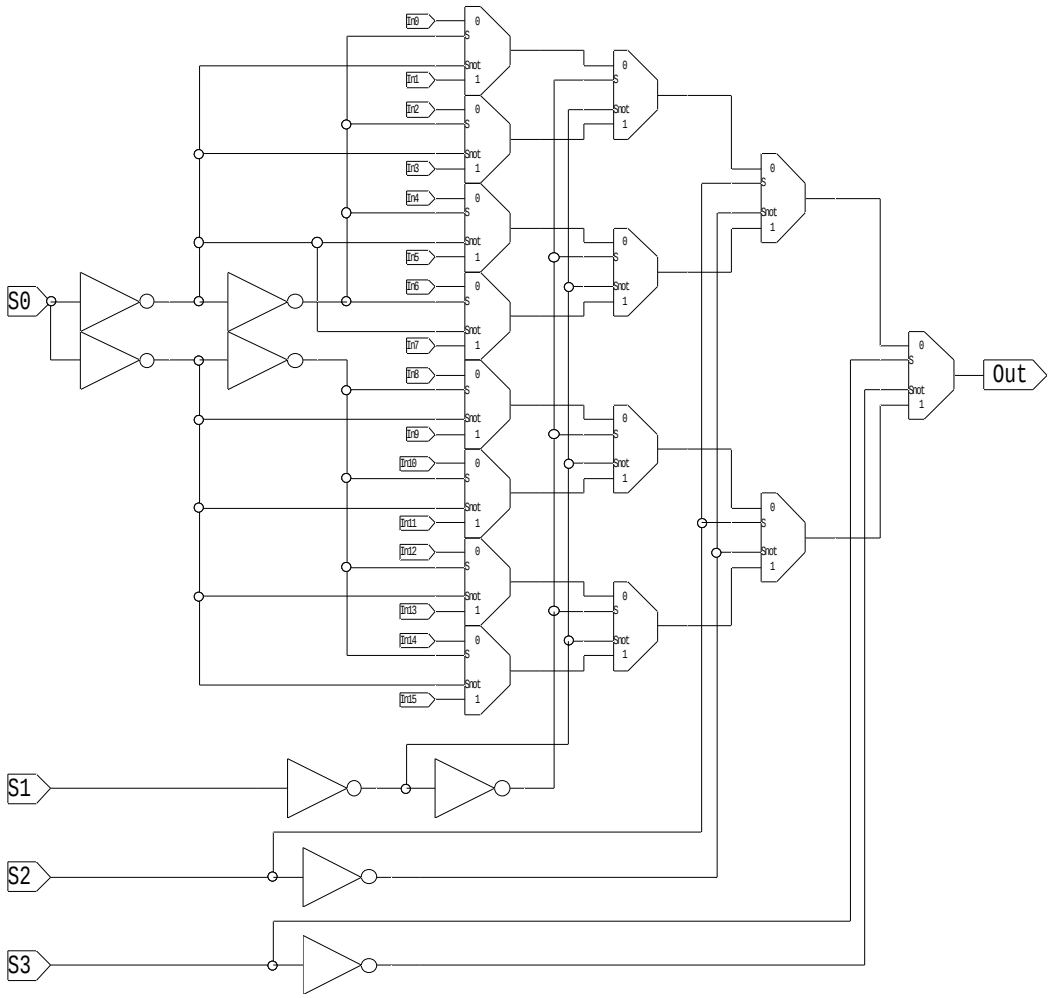

Figure B.12 Schematic of the 16-to-1 multiplexer circuit.

The multiplexer path delay analysis in Chapter II assumes that the change in delay through the variable delay module is due solely to the multiplexer select lines moving the multiplexer input tap along the delay chain. However, this assumption is not completely accurate. Detailed analysis of the path delay through the 16x1 multiplexer revealed that the delay was not constant, but rather varied by as much as 80 picoseconds between paths. Figure B.13 provides a table showing the delay through the 16x1 multiplexer for each multiplexer delay line tap.

Decimal Select Value	Multiplexer Delay (ns)	Mux Delay Difference (ns)	Incremental Delay (ns)
0	3.20	0.00	0.00
1	3.76	0.56	0.56
2	4.36	1.16	0.60
3	4.95	1.75	0.59
4	5.62	2.42	0.67
5	6.17	2.97	0.55
6	6.77	3.57	0.60
7	7.37	4.17	0.60
8	7.95	4.75	0.58
9	8.55	5.35	0.60
10	9.15	5.95	0.60
11	9.74	6.54	0.59
12	10.39	7.19	0.65
13	10.97	7.77	0.58
14	11.57	8.37	0.60
15	12.18	8.98	0.61

Figure B.13 Table of 16x1 multiplexer delay.

Figure B.14 is a plot of the data in the third column of Figure B.13, which shows the incremental delay of the multiplexer and inverter chain. Ideally, this graph should be a horizontal line corresponding to the value of delay attributed to two minimum-sized inverters, which was approximately 600 ps for this technology. However, the structure of the 16x1 multiplexer imposes a variable delay on the output signal that is a function of the path through the multiplexer. Fortunately, the use of a multiplexer for the delay-adjustment module was for proof-of-concept purposes only. A more practical application utilizing the proposed clock synchronization technique would most likely make use of a 5-to-32 decoder connected to 32 identical 2-to-1 multiplexers located in the DIS range bins. In this case, the delay increments would be approximately equal and would not show the variations portrayed in Figure B.14

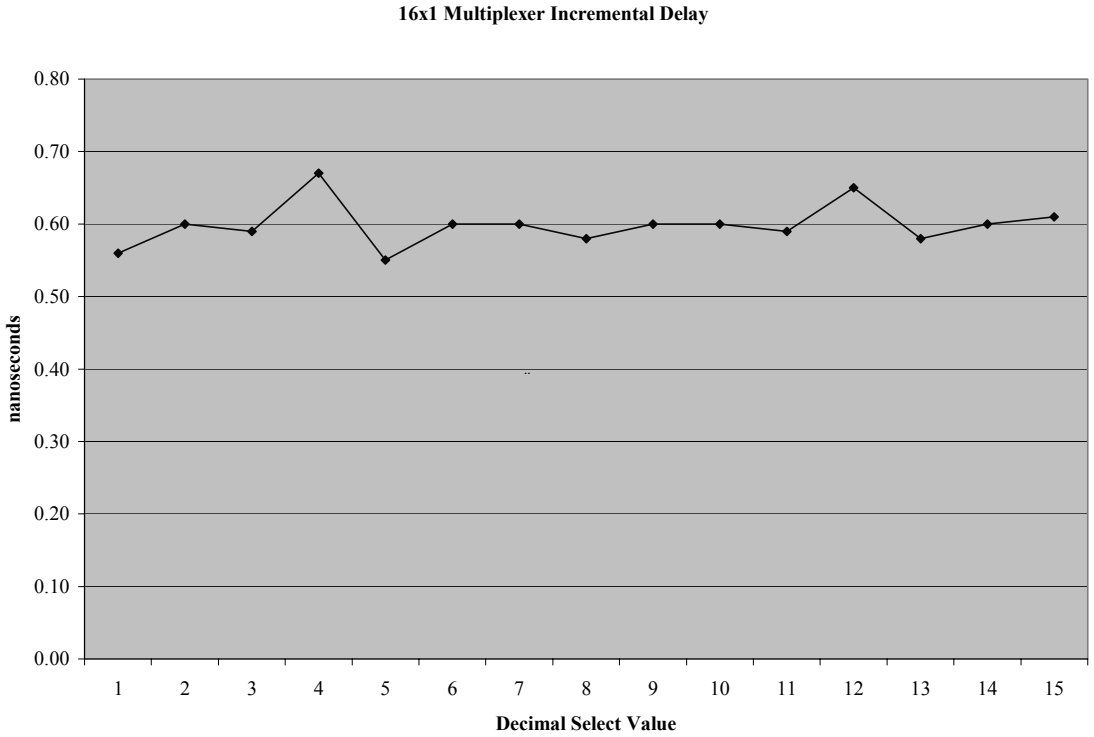

Figure B.14 Graph of 16x1 multiplexer delay.

Figure B.15 shows the power requirements for all modules larger than the basic logic gates. A graph of the data is provided in Figure B.16.

Logic Block	**PWR (W)**
Mux2x1	3.31E-03
DFF	5.60E-03
TFF	7.57E-03
Mux16x1	1.06E-02
Phase Check Module	1.80E-02
Finite State Machine	2.10E-02
Five bit counter	3.05E-02
Wraparound Counter	9.87E-02
Variable Delay Module	1.77E-01
Clock Sync Chip	5.36E-01
Clock Sync Chip and Pad Ring	2.18E+01

Figure B.15 Table of large circuit power.

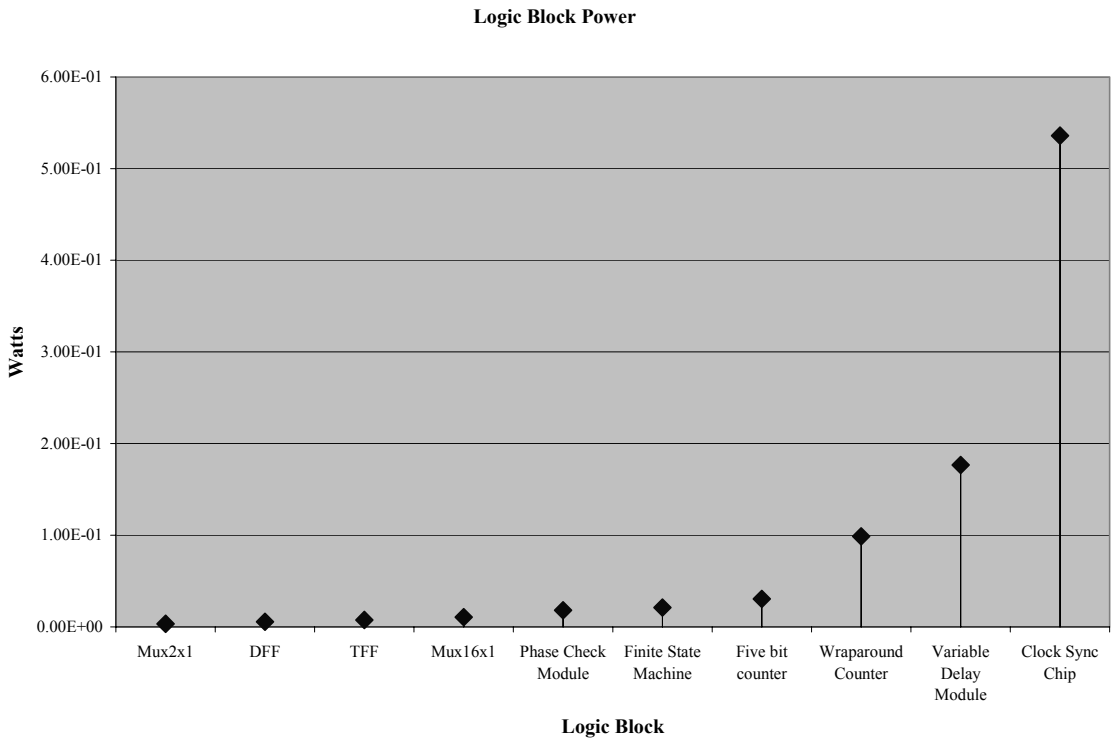

Figure B.16 Graph of large circuit power.

APPENDIX C RSNS MATLAB AND VISUAL BASIC CODE

The first section of this Appendix provides the MATLAB code for the *N*-modulus $\hat{M}$ search algorithm and the MATLAB code for the Pace/Styer $\hat{M}$ search program. The second section of this Appendix provides the MATLAB and Microsoft Visual Basic code used to convert the RSNS redundancy equations into the ArcGIS RSNS circle plot format. Figure C.1 shows the flow of the *SmartSearch* $\hat{M}$ search program.

A. $\hat{M}$ SEARCH ALGORITHM CODE

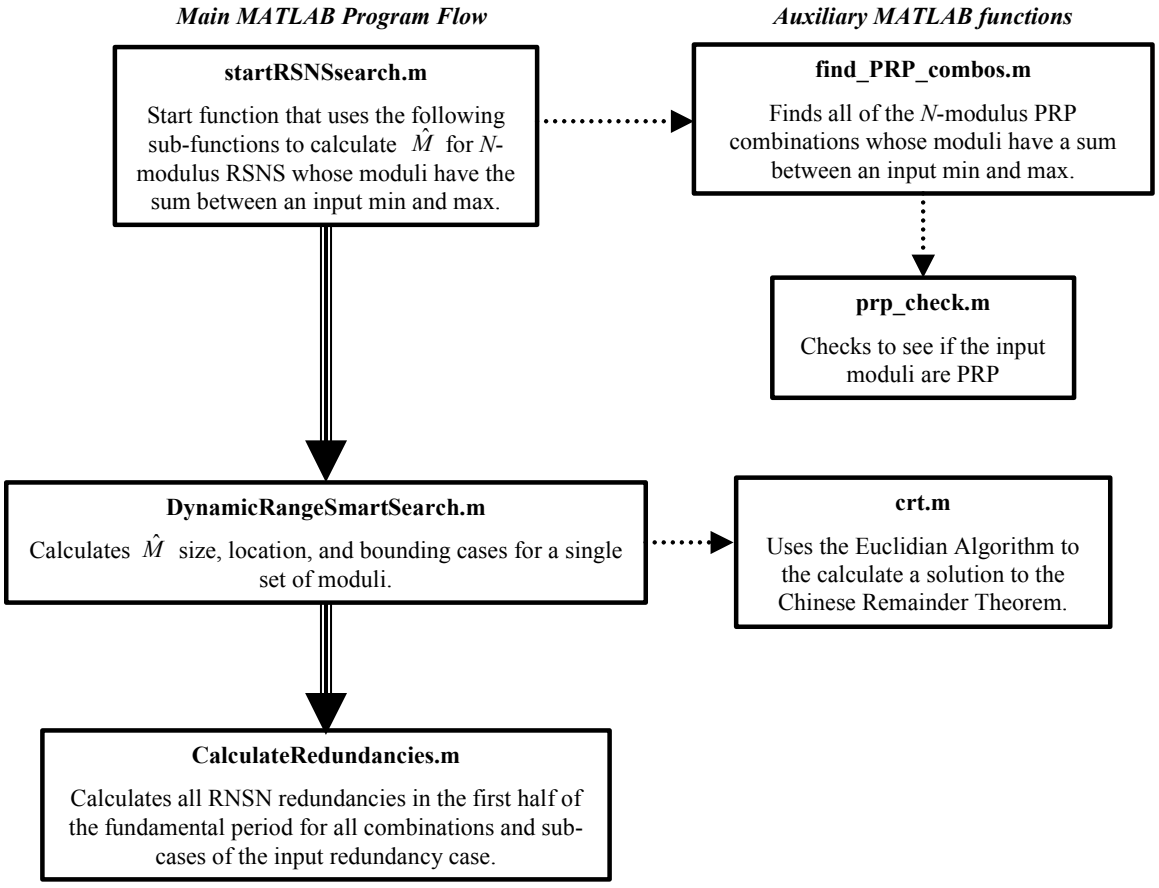

Figure C.1 Flowchart of $\hat{M}$ search algorithm code.

The following sections provide well-documented MATLAB code and sample program output for the *SmartSearch* program. The last section provides an $\hat{M}$ search program and sample output from Pace and Styer used for comparison to the *SmartSearch* program. Since these programs were written with an ADC application in mind, the term *dynamic range* is used synonymously with $\hat{M}$, *channel* is equivalent to MRS, and *MRSS* is equivalent to MRSS in the following programs.

1. Sample $\hat{M}$ Search Code Output

This *SmartSearch* sample MATLAB output shows $\hat{M}$ computation for all three-modulus PRP moduli whose sum is between 1 and 18.

```
>> num2str(startRSNSsearch(3,1,18,0))
Computing PRP combinations for 3 moduli that sum to 1 up to 18.
9 combinations remaining
Time = 0.00033333 minutes, 8 combinations remaining
Time = 0.00033333 minutes, 7 combinations remaining
Time = 0.00016667 minutes, 6 combinations remaining
Time = 0.00066667 minutes, 5 combinations remaining
Time = 0.00033333 minutes, 4 combinations remaining
Time = 0.00033333 minutes, 3 combinations remaining
Time = 0.00016667 minutes, 2 combinations remaining
Time = 0.00033333 minutes, 1 combinations remaining
Time = 0.00033333 minutes, 0 combinations remaining

ans =

12   3   4   5    360      79       220     121     312      43        0.02
14   3   4   7    504      17       231     67      232      51        0.02
15   3   5   7    630      207      312     266     220      60        0.04
16   3   5   8    720      278      211     338     230      61        0.02
16   4   5   7    840      -2       310     63      311      66        0.01
18   3   4   11   792      -2       310     68      110      71        0.01
18   3   4   11   792      -2       110     68      230      71        0.01
18   3   4   11   792      5        231     75      221      71        0.01
18   3   4   11   792      286      220     356     230      71        0.01
18   3   4   11   792      293      231     363     111      71        0.01
18   3   4   11   792      293      111     363     311      71        0.01
18   3   7   8    1008     196      220     275     230      80        0.02
18   4   5   9    1080     12       222     86      230      75        0.02
18   4   5   9    1080     132      312     206     230      75        0.02
18   5   6   7    1260     298      220     379     312      82        0.02

where the columns are:
sum  m1  m2  m3   Pf       start    case    stop    case     DR      time (s)
```

2. startRSNSsearch.m

```
function [MODLIST] = startRSNSsearch(Nc,SUM_MIN,SUM_MAX,savefile)

%**************************************************************************
%* StartRSNSsearch  Calculates all of the dynamic ranges for N-channel
%*     RSNS whose moduli have the sum between CMIN and CMAX, inclusive.
%*
%*     MODLIST = startRSNSsearch(Nc,SUM_MAX,SUM_MIN,0) returns a matrix of
%*     all combinations of N relatively prime moduli whose sum is between
%*     SUM_MIN and SUM_MAX.  The columns of the matrix are as follows:
%*     Moduli Sum, Moduli (N of them), Fundamental Period, DR lower bound,
%*     DR lower bound Case, DR upper bound, DR upper bound Case, Dynamic
%*     Range (DR), and run time.
%*
%*     If the savefile flag is zero, no files will be saved to disk.  If
%*     the savefile flag is not zero, a .mat and .txt file containing the
%*     matrix described above will be saved in the current directory
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 18AUG04
%*
%*     Called Functions: find_PRP_combos, DynamicRangeSmartSearch
%*     Calling Functions: None
%*
%**************************************************************************

global CMIN CMAX N Pf MODLIST COUNT COMBINATIONS MIN_DYNRANGE MAX_DYNRANGE

% Assigning global variables
N = Nc;
CMAX = SUM_MAX;
CMIN = SUM_MIN;

% This variable holds the result of the dynamic range search for all moduli
% combinations
MODLIST = [];

% This variable holds all of the pairwise relatively prime (PRP) moduli
% combinations
COMBINATIONS = [];

% This variable holds the count of the number of PRP combinations
COUNT = 0;

disp(['Computing PRP combinations for ',num2str(N), ...
        ' moduli that sum to ',num2str(CMIN),' up to ',num2str(CMAX),'.'])

% This loop finds all PRP moduli combinations of size N with a sum from
% CMIN to CMAX starting from (3 4 5 ... N+2)

for i = N:-1:1
    mods = 2+[1:N];
    find_PRP_combos(i,mods);
end

disp([num2str(COUNT),' combinations remaining'])

% This loop finds the dynamic range for each PRP moduli combination and
% adds it as a row to the MODLIST matrix.
```

```
% There may be more than one dynamic range for each moduli combination and
% there will be a separate row in the
% MODLIST matrix for each dynamic range found.

for i = 1:size(COMBINATIONS,1)
    mods = COMBINATIONS(i,:);
    tic                                     % start the MATLAB timer
    result = DynamicRangeSmartSearch(mods); % main search function
    t = toc;                                % stop the MATLAB timer
    s = size(result,1);

    % This loop adds all of the dynamic ranges to MODLIST for each set of
    % moduli
    for ii = 1:s
        MODLIST = [MODLIST; [sum(mods) mods 2*N*prod(mods) ...
                             result(ii,:) t]];
    end

    COUNT = COUNT - 1;
    disp(['Time = ',num2str(t/60),' minutes, ',num2str(COUNT), ...
            ' combinations remaining'])
end

MODLIST = sortrows(MODLIST,1);

% These statements save the data to disk in .mat and .txt format
if savefile ~= 0
    total_time_desktop = sum(MODLIST(:,size(MODLIST,2)))
    filename = [num2str(N),'Chan',num2str(CMIN),'to',...
                num2str(CMAX),'_desk'];
    eval(['save ',filename]);
    eval(['save ',filename,'.txt MODLIST -ASCII -TABS']);
end
```

3. DynamicRangeSmartSearch.m

```
function [dynamic_range] = DynamicRangeSmartSearch(modli)

%*********************************************************************
%* DynamicRangeSmartSearch  Calculates the dynamic ranges for input set of
%*     moduli. dynamic_range = DynamicRangeSmartSearch(moduli) returns
%*     a matrix whose rows contain information about the of all dynamic
%*     ranges for the input set of moduli. The columns of the matrix
%*     are as follows: Moduli Sum, Moduli (N of them), Fundamental Period,
%*     DR lower bound, DR lower bound Case, DR upper bound, DR upper
%*     bound Case, and Dynamic Range (DR).
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 19AUG04
%*
%*     Called Functions: crt, CalculateRedundancies
%*     Calling Functions: startRSNSsearch
%*
%*********************************************************************

global CMIN CMAX N Pf MODLIST COUNT COMBINATIONS MIN_DYNRANGE MAX_DYNRANGE

dyn_ranges = [];
comb_dyn_ranges = [];
```

```
intervals = [];

% Fundamental period for PRP moduli
Pf = 2*N*prod(modli);

% This loop forms the circularly shifted and linearly increasing NxN
% channel matrix (chan). Channels are in the rows and SUB-CHANNELs are in
% the columns
chan = [];
for i = 1:N
    hshift = [0:-1:-N+1]+i-1;
    if i > 1
        hshift = [hshift(i:max(size(hshift))) hshift(1:i-1)];
    end
    chan(i,:) = hshift;
end

% This loop computes binomial coefficients to determine how many
% combinations are in each case.  The result put into a 2^N x N
% matrix of 1's and 0's called sortedBin
binChar = dec2bin([0:2^N-1]',N);
pad = repmat(' ',max(size(binChar)),1);
for i = 1:N
    binChar_pad(:,2*i) = pad;
    binChar_pad(:,2*i-1) = binChar(:,i);
end
binNum = str2num(char(binChar_pad));
binNum(:,N+1) = sum(binNum,2);
sortedBin = sortrows(binNum,[N+1 1:N]);

% This loop uses the CRT to find all Case N1X COR, which are the
% fundamental COR shifts for all other cases and sub-cases
% The vector holding the COR is called COR, and the vector redundancyID
% contains the case label of the corresponding COR
COR = [];
redundancyID = [];
for subchan = 0:N-1
    ch = chan(:,subchan+1)';
    m = max(ch);
    a = (m-ch)/N;
    center_of_redundancy = crt(a,modli)*N-m;
    if center_of_redundancy < 0
        center_of_redundancy = center_of_redundancy + N*prod(modli);
    end
    COR = [COR center_of_redundancy];
    redundancyID = [redundancyID ...
            str2num([num2str(N),'1',num2str(subchan)])];
end

% This loop computes the Case N1X redundancies and adds them to the vector
% called dyn_ranges, which will be used to compute the zones of maximum
% possible dynamic ranges in which to search for the dynamic range
[CORtemp,index] = sort(COR);
CORtemp = [CORtemp Pf/2];
redundancyID = [redundancyID(index) str2num([num2str(N),'10'])];
for i = 1:length(CORtemp)-1
    dyn_ranges = [dyn_ranges; [CORtemp(i)-N+1  redundancyID(i) ...
                CORtemp(i+1)+N-1 redundancyID(i+1) ...
                CORtemp(i+1)+2*N-CORtemp(i)-1]];
end

% The following statements set the maximum and minimum dynamic range based
```

```
% on the current moduli set and the fact that the minimum dyanamic range
% for an n-channel RSNS is always larger than the minimum dynamic range for
% an (n-1)-channel RSNS.
dyn_ranges_loop = flipud(sortrows(dyn_ranges,5));
MAX_DYNRANGE = dyn_ranges_loop(1,5);
MIN_DYNRANGE_limits = [1 1 42 116 419 1615]; % values are from previous runs

if N > length(MIN_DYNRANGE_limits)
    MIN_DYNRANGE = MIN_DYNRANGE_limits(length(MIN_DYNRANGE_limits));
else
    MIN_DYNRANGE = MIN_DYNRANGE_limits(N);
end

% This loop calculates all redundancies for all cases by calling the
% function CalculateRedundancies and stores them in a matrix
redundancy_vector = [];
for i = N:-1:1
    redundancy_vector = [redundancy_vector; ...
            CalculateRedundancies(COR,sortedBin,modli,i)];
end

% These statements sort the redundancies and prepare the redundancy vector
% for searching
temp_vector = sortrows(redundancy_vector,[2 1]);
[B,I,J] = unique(temp_vector(:,2));
redundancy_vector = temp_vector(I,:);
redundancy_vector = [redundancy_vector [1:length(redundancy_vector)]'];

% This next section searches through the matrix of redundancies to find
% consecutive redundancies.  Once consecutive redundancies are found, the
% number of vectors between the endpoints of the redundancies is a
% potential dynamic range.  The largest string of such vectors is the
% dynamic range.  The search routine finds all intervals and then chooses
% the largest one(s) as the dynamic range.

% find the first redundancy
interval_start = redundancy_vector(1,:);
len = size(redundancy_vector,1);
start_pointer = 2;

% compute the starting position of all redundancies (all_starts) and the
% ending position of all redundancies (all_stops) for redundancies that are
% smaller than the maximum dynamic range
max_dynrange2 = MAX_DYNRANGE + 2;
all_stops = find(redundancy_vector(start_pointer:len,3) <= ...
    (interval_start(1)+max_dynrange2));
stop_vector = redundancy_vector(all_stops+start_pointer-1,:);
all_starts = find(stop_vector(:,1) >= interval_start(1));
intervals = [interval_start];

% loop through all of the redundancies and compute the distance bewteen the
% start and end points of consecutive redundancies (intervals)
% which are all potential dynamic ranges
while isempty(all_stops)~= 1
    sorted_stops = sortrows(stop_vector(all_starts,:),3);
    interval_stop = sorted_stops(1,:);
    intervals = [intervals; interval_stop];
    interval_start = interval_stop;
    find_stop = find(stop_vector(:,5) == interval_stop(5));
    start_pointer = start_pointer+find_stop;
    all_stops = find(redundancy_vector(start_pointer:len,3) <= ...
        (interval_start(1)+max_dynrange2));
```

```
    stop_vector = redundancy_vector(all_stops+start_pointer-1,:);
    all_starts = find(stop_vector(:,1) >= interval_start(1));
end

% This section computes the sizes of the intervals and keeps only the
% largest intervals which are the dynamic ranges
len = length(intervals);
interval_sizes = [intervals(1:len-1,1)+1 intervals(1:len-1,4) ...
        intervals(2:len,3)-1 intervals(2:len,4)];
interval_sizes(:,5) = interval_sizes(:,3)-interval_sizes(:,1)+1;

dynamic_range = interval_sizes(find(interval_sizes(:,5)== ...
    max(interval_sizes(:,5))),:);
```

4. CalculateRedundancies.m

```
function [redundancies] = CalculateRedundancies(cor,sortedBin,modli,CaseID)

%*************************************************************************
%* CalculateRedundancies  Calculates all RNSN redundancies in the first
%*     half of the fundamental period for the all combinations of the input
%*     CaseID.
%*
%*     redundancies = CalculateRedundancies(cor,sortedBin,modli,CaseID)
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 18AUG04
%*
%*     Called Functions: None
%*     Calling Functions: DynamicRangeSmartSearch
%*
%*************************************************************************

global CMIN CMAX N Pf MODLIST COUNT COMBINATIONS MAX_DYNRANGE MIN_DYNRANGE

redundancies = [];

% Finds all of the combinations that match the CaseID and stores them in a
% matrix called allBinaryNums which has 1's and 0's in each row that
% represents a particular of Type 1 and Type 0 redundancies.  Each row
% represents an individual combination of Type 1 and Type 0 redundancies
% for the particular case.
[rows,cols] = find(sortedBin(:,min(size(sortedBin))) == CaseID);
allBinaryNums = sortedBin(rows,1:N);
s = size(allBinaryNums,1);

% This loop goes through each combination and computes and labels the
% redundancies from 0 to Pf/2 for every SUB-CHANNEL.
for ii = 1:s

    %form the string identifier for this particular case
    strID = [num2str(CaseID),num2str(ii)];

    % find all Type 1 and Type 0 redundancies
    f1 = find(allBinaryNums(ii,:)==1);
    f0 = find(allBinaryNums(ii,:)==0);

    % computing partial results for the length and COR of the redundancy
    m = prod(modli(f0));
```

```
    mm = prod(modli(f1));

    % redundancy length
    redundancy_len = 2*N*m;

    % continue only if the redundancy will affect the maximum dynamic range
    if redundancy_len <= MAX_DYNRANGE

        % compute all COR and labels for the 0th subchannel
        CORvector = [0:N*mm:Pf-1];
        numID = str2num([strID,'0']);
        tempCORvector = [CORvector' repmat(numID,length(CORvector),1)];

        % compute all COR and labels for all other SUB-CHANNELs
        for sub_chan = 1:N-1
            numID = str2num([strID,num2str(sub_chan)]);
            tempCORvector_shift = [CORvector'+cor(sub_chan+1) ...
                    repmat(numID,length(CORvector),1)];
            tempCORvector = [tempCORvector; tempCORvector_shift];
        end

        % adjust COR so that they are all in a single Pf
        f = find(tempCORvector(:,1)>Pf);
        tempCORvector(f,1) = tempCORvector(f,1)-Pf;

        % Only keep COR located in 0 to Pf/2 since the rest are redundant
        f = find(tempCORvector(:,1)<=Pf/2);
        tempCORvector = tempCORvector(f,:);

        % compute start and stop endpoints in a vector of the form
        % [StartPoint COR StopPoint CaseLabel]
        startstop_vector = [tempCORvector(:,1)-N*m tempCORvector(:,1) ...
                tempCORvector(:,1)+N*m tempCORvector(:,2)];
        redundancies = [redundancies; startstop_vector];

        % compute the maximum dynamic range for this case and if it is
        % smaller than the previous maximum, replace with current max.
        case_max_dynrange = N*mm+redundancy_len-1;
        if MAX_DYNRANGE > case_max_dynrange
            MAX_DYNRANGE = case_max_dynrange;
        end
    end
end
```

5. find_PRP_combos.m

```
function [pos,mods] = find_PRP_combos(pos,mods)

%**********************************************************************
% find_PRP_combos  Finds all of the PRP combinations starting from
%     the moduli mods and incrementing all moduli above position pos
%     until the sum exceeds the global variable CMAX. The PRP
%     combinations are stored in the global variable COMBINATIONS
%     and the PRP combinations count is stored in COUNT.  The function
%     calls itself recursively to search through all possible moduli.
%
%     Usage: [pos,mods] = find_PRP_combos(pos,mods)
%
%     Author:        LCDR Brian Luke
```

```
%     Last Modified: 18AUG04
%
%     Called Functions: prp_check, find_PRP_combos (recursive)
%     Calling Functions: startRSNSsearch
%
%*********************************************************************

global CMIN CMAX N MODLIST COUNT COMBINATIONS

% This function takes a set of moduli and a pointer at a single modulus.
% If pointer is pointing at the last modulus, then the function increments
% the last modulus until the maximum sum is reached and checks each set of
% moduli created for PRP.  If the pointer is not pointed at the last
% modulus, the function increments the modulus at the pointer, increments
% the pointer, and calls itself recursively.

if pos == N
    while (sum(mods) <= CMAX)
        if (prp_check(mods) == 1) & (sum(mods) >= CMIN)
            COUNT = COUNT + 1;
            COMBINATIONS = [COMBINATIONS; mods];
        end
        mods(N) = mods(N)+1;
    end

else
    temp_mods = mods;
    temp_mods(pos) = mods(pos)+1;

    while (sum(temp_mods) <= CMAX)
        [n,mods] = find_PRP_combos(pos+1,temp_mods);
        temp_mods = mods;
        temp_mods(pos) = mods(pos)+1;
        temp_mods(pos+1:N) = mods(pos)+[1:N-pos];
    end
end
```

6. prp_check.m

```
function [prp] = prp_check(mods)

%*****************************************************************
% prp_check  Checks to see if the moduli in mods are PRP.  If yes,
%     then the prp flag set to 1, otherwise 0.  The function
%     accomplishes this by finding all factors of all moduli and
%     compares this to the unique factors of all moduli.  If
%     they are not the same, then there are duplicate factors
%     among the moduli and they are not prp.
%
%     Usage: [prp] = prp_check(mods)
%
%     Author:        LCDR Brian Luke
%     Last Modified: 18AUG04
%
%     Called Functions: None
%     Calling Functions: find_PRP_combos
%
%*****************************************************************
```

```
global N

factors = [];
for i = 1:N
    factors = [factors unique(factor(mods(i)))];
end

f = unique(factors);

if size(f) == size(factors)
    prp = 1;
else
    prp = 0;
end
```

7. crt.m

```
function [x] = crt(a,moduli)
%*************************************************************************
% crt  This function uses the Euclidian Algorithm to the calculate
%      a solution to the Chinese Remainder Theorem.  The solution is of the
%      form x = sum([M/m1 M/m2 ...].*[a1 a2 ...].*[b1 b2 ...]) where
%      M=prod([m1 m2 ...]) and [b1 b2 ...] come from the Euclidian
%      Algorithm.
%
%     Usage: x = crt([a1 a2 a3 a4 ...],[m1 m2 m3 m4 ...])
%
%     Author:        LCDR Brian Luke
%     Last Modified: 19AUG04
%
%     Called Functions: None
%     Calling Functions: find_PRP_combos
%
%*************************************************************************

M = prod(moduli);      % product of moduli
Mmi = [];              % This vector holds the M/mi values
b = zeros(size(a));    % This vector holds all of the solution coefficients

for i = 1:max(size(moduli))

    % Initialize Euclidian Algorithm:
    % M/mi(1) + mi(0) = M/mi
    % M/mi(0) + mi(1) = mi
    %
    % where the c matrix contains the coefficients
    % in parenthesis in the above equations
    c = [ 1 0; 0 1];
    mi = moduli(i);
    Mm = [M/mi mi];

    % Loop until M/mi(c21) + mi(c22) = 1
    while Mm*c(2,:)' ~= 1
        r0 = Mm*c(1,:)';
        r1 = Mm*c(2,:)';
        r = floor(r0/r1);
        c_new(2,:) = c(1,:) - c(2,:)*r;
        c_new(1,:) = c(2,:);
        c = c_new;
```

```
    end

    % Save the modulus and the coefficient
    Mmi = [Mmi M/mi];
    b(i) = c(2,1);
end

% Compute the solution to the CRT using the coefficients from the Euclidian
% Algorithm and adjust to choose the least positive solution.
x = sum(Mmi.*a.*b);
if x < 0
    x = x + M*(floor(abs(x/M))+1);
elseif x >= M
    x = x - M*(floor(abs(x/M)));
end
```

8. PaceStyerRSNSsearch.m

This program is the $\hat{M}$ search program written by Pace and Styer used for comparison to the *Smartsearch* search program.

```
% THIS PROGRAM FINDS MAXIMAL STRINGS OF NON-REDUNDANT
% VECTORS FOR THE N (N >2 AND ANY INTEGER) CHANNEL ROBUST SNS.

clear all

% DEFINE THE CHANNEL NUMBER,THE VALUES OF CHANNELS AND
% THE NUMBER OF SEARCH WRT FUNDEMENTAL PERIOD

disp('This program finds the maximal strings of non-redundant vectors');
disp('for the N channel ROBUST SNS');
chanum=input('Enter the Number of Channels for ROBUST SNS >> ');

M=1;
for i=1:chanum
    m(i)=input(['Enter ' int2str(i) '.Channel Value >> ']);
    M=M*m(i);
end
period=2*chanum*M;
nsearch=period+10;

prompt='y';

% DEFINE THE SHIFT AMOUNT OF CHANNELS
while (prompt=='y')|(prompt=='Y')

    for i=1:chanum
        s(i)=input(['Enter ' int2str(i) '. Channel Shift Value >> ']);
    end

    % INITIATE THE VARIABLES TO ZERO
    i=0;
    ii=0;
    j=0;
    jj=0;
```

```
k=0;
funper=0;
dynrange=0;

% DEFINE THE WAVEFORM FOR CHANNEL m(1),m(2)...m(n) IN THE FORM OF MATRIX g

for r=1:chanum
    mm(r,[1 2])=[m(r) s(r)];

    for i=1+s(r):chanum*m(r)+s(r)
        g(r,i)=floor((i-s(r))/chanum);
    end

    for i=chanum*m(r)+1+s(r): 2*chanum*m(r)+s(r)
        g(r,i)=floor((2*chanum*m(r)+chanum-i+s(r)-1)/chanum);
    end

    g(r,2*chanum*m(r)+s(r)+1:4*chanum*m(r)+s(r))=...
        g(r,1+s(r):2*chanum*m(r)+s(r));
    g(r,4*chanum*m(r)+s(r)+1:8*chanum*m(r)+s(r))=...
        g(r,1+s(r):4*chanum*m(r)+s(r));
    g(r,8*chanum*m(r)+s(r)+1:16*chanum*m(r)+s(r))=...
        g(r,1+s(r):8*chanum*m(r)+s(r));
    g(r,16*chanum*m(r)+s(r)+1:32*chanum*m(r)+s(r))=...
        g(r,1+s(r):16*chanum*m(r)+s(r));
    g(r,32*chanum*m(r)+s(r)+1:64*chanum*m(r)+s(r))=...
        g(r,1+s(r):32*chanum*m(r)+s(r));
    g(r,64*chanum*m(r)+s(r)+1:128*chanum*m(r)+s(r))=...
        g(r,1+s(r):64*chanum*m(r)+s(r));
    g(r,128*chanum*m(r)+s(r)+1:256*chanum*m(r)+s(r))=...
        g(r,1+s(r):128*chanum*m(r)+s(r));
    g(r,256*chanum*m(r)+s(r)+1:512*chanum*m(r)+s(r))=...
        g(r,1+s(r):256*chanum*m(r)+s(r));
end

% DEFINE MATRIX ga AND THIS MATRIX IS THE TRANSPOSE OF MATRIX g

ga=g';

% DEFINE MATRIX gb AND THIS MATRIX GIVES THE ROW NUMBER IN COLUMN 1,
% AND THE VECTOR FOR THAT ROW IN COLUMNS 2 THROUGH THE NUMBER OF CHANNEL
% DEFINED IN MATRIX ga

gb(:,[2:(chanum+1)])=ga(:,[1:chanum]);
[sgbr,sgbc]=size(gb);
gb(:,1)=(1:1:sgbr)';

% DEFINE THE MATRIX gc.THIS MATRIX GIVES THE ROW NUMBER THROUGH
% THE NUMBER OF SEARCH (eliminate the parts of the matrix beyond
% the search length)

gc=gb;
gc(nsearch+1:sgbr,:)=[];
[sgcr,sgcc]=size(gc);

% FIND THE FIRST REDUNDANCIES IN MATRIX gc

k=1;
for ii=2:nsearch; % ii is row index into gc
    xrec=gc(ii,[2:(chanum+1)]);
    for jj=ii+1:sgcr;
        if gc(jj,[2:(chanum+1)])==xrec
```

```
            redun=gc(jj,1);

            % DEFINE THE MATRIX h WHICH IS THE MATRIX OF
            % FIRST REDUNDANCIES

            h(k,1)=ii;
            h(k,2)=redun;
            k=k+1;
            break
        end
    end
end

% DEFINE THE MATRIX hsort AND SORT BY THE REDUNDANCY COLUMN IN MATRIX h

hsort=h;
[yoy,ioi]=sort(hsort);

% DEFINE THE MATRIX hsorted

hsorted=[yoy(ioi(:,2),1) yoy(:,2)];
hsorted;

% DEFINE THE MATRIX hreduced.
% ELIMINATE THE ROWS OF THE MATRIX hsorted THAT DO NOT ALLOW
% THE FIRST COLUMN TO BE MONOTONE INCREASING
% ssr - rows of hsorted
% ssc - columns of hsorted
% a - value in last row of h (hsort)
% rx - rows in hreduced
% cx - columns in hreduced

[ssr,ssc]=size(hsorted);
hreduced=hsorted;
a=hsort(ssr,1);
[rx cx]=size(hreduced);
for k=1:ssr
    for i=1:ssr
        if i<rx
            if hreduced(i,1)==a
                hreduced(i+1:rx,:)=[];
                break
            elseif hreduced(i+1,1)<hreduced(i,1)
                hreduced(i+1,:)=[];
                break
            end
        end
        [rx cx]=size(hreduced);
    end
end
hreduced;

% DEFINE THE MATRIX H THAT SHOWS WHICH SETS OF ROWS
% ARE MAXIMAL FOR NO REDUNDANCIES AND THEIR LENGTHS.

[hsr,hsc]=size(hreduced);
H(1,1)=(chanum-1);
H(2:hsr+1,1)=hreduced(1:hsr,1)+1;
H(1:hsr,2)=hreduced(1:hsr,2)-1;
H(hsr+1,2)=nsearch;
H(1:hsr+1,3)=H(1:hsr+1,2)-H(1:hsr+1,1)+1;
```

```
    % FIND THE DYNAMIC RANGE OF N-CHANNEL RSNS

    HH=max(H);
    dynrange=HH(:,3);

    % DISPLAY A MATRIX THAT SHOWS THE BEGIN-END POSITION,
    % DYNAMIC RANGE AND ALSO CHANNEL-SHIFT VALUES

    disp(' ')
    disp(' ')
    disp(['THE FUNDEMANTAL PERIOD IS ',num2str(period),' '])
    disp(' ')
    disp(' ')
    disp(['THE DYNAMIC RANGE IS ',num2str(dynrange),' '])

    fprintf('\n    BEGIN POSITION  END POSITION  DYNAMIC RANGE\n')
    fprintf('    --------------  ------------  -------------\n')
    fprintf('%11.0f %16.0f %12.0f \n',H')


    fprintf('\n    CHANNEL VALUES     SHIFT VALUES\n')
    fprintf('    --------------     ------------\n')
    fprintf('%11.0f %17.0f \n',mm')


    prompt=input('Would you like to try another shift (y/n) ? >>','s');

end
```

9. Sample Program Output for Pace/Styer Search Program

```
This program finds the maximal strings of non-redundant vectors
for the N channel ROBUST SNS
Enter the Number of Channels for ROBUST SNS >> 3
Enter 1.Channel Value >> 3
Enter 2.Channel Value >> 4
Enter 3.Channel Value >> 5
Enter 1. Channel Shift Value >> 0
Enter 2. Channel Shift Value >> 1
Enter 3. Channel Shift Value >> 2


THE FUNDEMANTAL PERIOD IS 360


THE DYNAMIC RANGE IS 43

    BEGIN POSITION  END POSITION  DYNAMIC RANGE
    --------------  ------------  -------------
          2               29           28
          7               30           24
          8               41           34
         13               51           39
         23               52           30
         24               57           34
         41               65           25
         61              103           43
         81              111           31
        107              130           24
```

```
        114             131          18
        115             148          34
        126             159          34
        131             160          30
        132             164          33
        142             165          24
        143             177          35
        161             184          24
        180             209          30
        187             210          24
        188             221          34
        193             231          39
        203             232          30
        204             237          34
        221             245          25
        241             283          43
        261             291          31
        287             310          24
        294             311          18
        295             328          34
        306             339          34
        311             340          30
        312             344          33
        322             345          24
        323             357          35
        341             364          24
        360             370          11

    CHANNEL VALUES     SHIFT VALUES
    --------------     ------------
          3                  0
          4                  1
          5                  2
Would you like to try another shift (y/n) ? >>n
```

B. ARCGIS SHAPEFILE GENERATION CODE

This section provides the MATLAB and Microsoft Visual Basic code used to convert the RSNS redundancy equations into the ArcGIS RSNS circle plot format. Figure C.2 provides the program flow that converts the MATLAB-generated RSNS redundancy locations to the ArcGIS shapefile format.

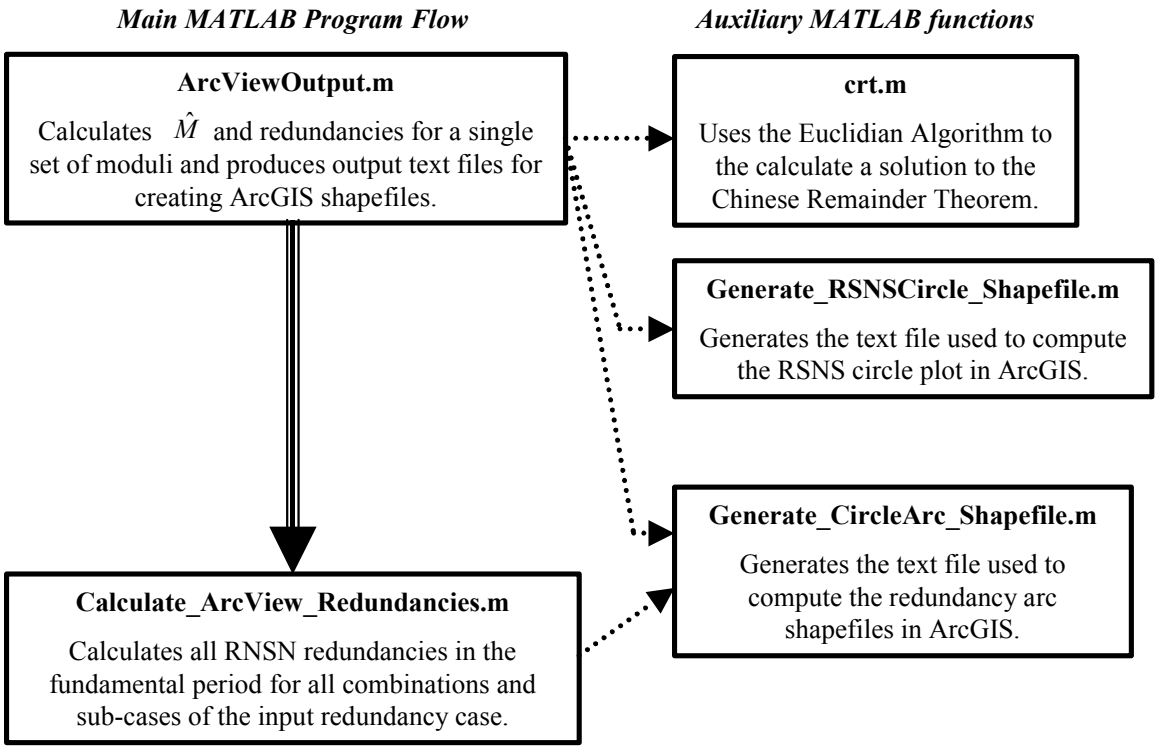

Figure C.2 Flowchart of MATLAB ArcGIS shapefile generation code.

1. Sample Shapefile Generation Code Output

The sample output provided in this section is a comma-delimited text file that is converted to a geographic shapefile in ArcGIS using the Visual Basic code in this Appendix.

```
Lon,Lat,h_mod_N,h,RSNS
0,0,0,0,New
5.00000000000000,0.00000000000000,0,0,"0  0  0"
4.99923847578196,0.08726203218642,0,0,"0  0  0"
4.91198530428718,0.08573902241278,0,0,"0  0  0"
4.91273353740028,0.00000000000000,0,0,"0  0  0"
5.00000000000000,0.00000000000000,0,0,"0  0  0"
0,0,0,0,New
4.99923847578196,0.08726203218642,1,1,"0  0  1"
4.99695413509548,0.17449748351250,1,1,"0  0  1"
4.90974083286692,0.17145192788877,1,1,"0  0  1"
4.91198530428718,0.08573902241278,1,1,"0  0  1"
4.99923847578196,0.08726203218642,1,1,"0  0  1"
0,0,0,0,New
4.99695413509548,0.17449748351250,2,2,"0  1  1"
4.99314767377287,0.26167978121472,2,2,"0  1  1"
4.90600080682724,0.25711260744662,2,2,"0  1  1"
```

```
4.90974083286692,0.17145192788877,2,2,"0  1  1"
4.99695413509548,0.17449748351250,2,2,"0  1  1"
```

2. ArcViewOutput.m

```
function [dynamic_range] = ArcViewOutput(modli)

%**************************************************************************
%* ArcViewOutput  Calculates the dynamic ranges for input set of
%*     moduli and generates the text files that ArcGIS uses to
%*     generate shapefiles. dynamic_range = ArcViewOutput(moduli) returns
%*     a matrix whose rows contain information about the of all dynamic
%*     ranges for the input set of moduli. The columns of the matrix
%*     are as follows: Moduli Sum, Moduli (N of them), Fundamental Period,
%*     DR lower bound, DR lower bound Case, DR upper bound, DR upper
%*     bound Case, and Dynamic Range (DR).
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 19AUG04
%*
%*     Called Functions:  Generate_RSNSCircle_Shapefile,
%                         Calculate_ArcView_Redundancies, crt
%*     Calling Functions: None
%*
%**************************************************************************

global N Pf MIN_DYNRANGE MAX_DYNRANGE RADIUS

dyn_ranges = [];
comb_dyn_ranges = [];
intervals = [];

% Fundamental period for PRP moduli
N = max(size(modli));
Pf = 2*N*prod(modli);

% Set the radius of the RSNS circle
RADIUS = 5;

% generate RSNS circle plot filename and shapefile
fname = [num2str(N),'Channel'];
for i = 1:N
    fname = [fname,'_',num2str(modli(i))];
end
fname = [fname,'_CirclePlot'];
Generate_RSNSCircle_Shapefile(modli,fname);

% This loop forms the circularly shifted and linearly increasing NxN
% channel matrix (chan). Channels are in the rows and SUB-CHANNELs are in
% the columns
chan = [];
for i = 1:N
   hshift = [0:-1:-N+1]+i-1;
   if i > 1
      hshift = [hshift(i:max(size(hshift))) hshift(1:i-1)];
   end
   chan(i,:) = hshift;
end
```

```
% This loop computes binomial coefficients to determine how many
% combinations are in each case.  The result put into a 2^N x N
% matrix of 1's and 0's called sortedBin
binChar = dec2bin([0:2^N-1]',N);
pad = repmat(' ',max(size(binChar)),1);
for i = 1:N
   binChar_pad(:,2*i) = pad;
   binChar_pad(:,2*i-1) = binChar(:,i);
end
binNum = str2num(char(binChar_pad));
binNum(:,N+1) = sum(binNum,2);
sortedBin = sortrows(binNum,[N+1 1:N]);

% This loop uses the CRT to find all Case N1X COR, which are the
% fundamental COR shifts for all other cases and sub-cases
% The vector holding the COR is called COR, and the vector redundancyID
% contains the case label of the corresponding COR
COR = [];
redundancyID = [];
for subchan = 0:N-1
   ch = chan(:,subchan+1)';
   m = max(ch);
   a = (m-ch)/N;
   center_of_redundancy = crt(a,modli)*N-m;
   if center_of_redundancy < 0
      center_of_redundancy = center_of_redundancy + N*prod(modli);
   end
   COR = [COR center_of_redundancy];
   redundancyID = [redundancyID ...
                   str2num([num2str(N),'1',num2str(subchan)])];
end

% This loop computes the Case N1X redundancies and adds them to the vector
% called dyn_ranges, which will be used to compute the zones of maximum
% possible dynamic ranges in which to search for the dynamic range
[CORtemp,index] = sort(COR);
CORtemp = [CORtemp Pf/2];
redundancyID = [redundancyID(index) str2num([num2str(N),'10'])];
for i = 1:length(CORtemp)-1
   dyn_ranges = [dyn_ranges; [CORtemp(i)-N+1  redundancyID(i) ...
                              CORtemp(i+1)+N-1 redundancyID(i+1) ...
                              CORtemp(i+1)+2*N-CORtemp(i)-1]];
end

% The following statements set the maximum and minimum dynamic range based
% on the current moduli set and the fact that the minimum dyanamic range
% for an n-channel RSNS is always larger than the minimum dynamic range for
% an (n-1)-channel RSNS.
dyn_ranges_loop = flipud(sortrows(dyn_ranges,5));
MAX_DYNRANGE = dyn_ranges_loop(1,5);
MIN_DYNRANGE_limits = [1 1 42 116 419 1615]; % values are from previous runs

if N > length(MIN_DYNRANGE_limits)
   MIN_DYNRANGE = MIN_DYNRANGE_limits(length(MIN_DYNRANGE_limits));
else
   MIN_DYNRANGE = MIN_DYNRANGE_limits(N);
end

% This loop calculates all redundancies for all cases by calling the
% function CalculateRedundancies and stores them in a matrix
redundancy_vector = [];
for i = N:-1:1
```

```
   redundancy_vector = [redundancy_vector; ...
                        Calculate_ArcView_Redundancies(COR,sortedBin,modli,i)];
end

% These statements sort the redundancies and prepare the redundancy vector
% for searching
temp_vector = sortrows(redundancy_vector,[2 1]);
[B,I,J] = unique(temp_vector(:,2));
redundancy_vector = temp_vector(I,:);
redundancy_vector = [redundancy_vector [1:length(redundancy_vector)]'];

% This next section searches through the matrix of redundancies to find
% consecutive redundancies.  Once consecutive redundancies are found, the
% number of vectors between the endpoints of the redundancies is a
% potential dynamic range.  The largest string of such vectors is the
% dynamic range.  The search routine finds all intervals and then chooses
% the largest one(s) as the dynamic range.

% find the first redundancy
interval_start = redundancy_vector(1,:);
len = size(redundancy_vector,1);
start_pointer = 2;

% compute the starting position of all redundancies (all_starts) and the
% ending position of all redundancies (all_stops) for redundancies that are
% smaller than the maximum dynamic range
max_dynrange2 = MAX_DYNRANGE + 2;
all_stops = find(redundancy_vector(start_pointer:len,3) <= ...
                 (interval_start(1)+max_dynrange2));
stop_vector = redundancy_vector(all_stops+start_pointer-1,:);
all_starts = find(stop_vector(:,1) >= interval_start(1));
intervals = [interval_start];

% loop through all of the redundancies and compute the distance bewteen the
% start and end points of consecutive redundancies (intervals)
% which are all potential dynamic ranges
while isempty(all_stops)~= 1
   sorted_stops = sortrows(stop_vector(all_starts,:),3);
   interval_stop = sorted_stops(1,:);
   intervals = [intervals; interval_stop];
   interval_start = interval_stop;
   find_stop = find(stop_vector(:,5) == interval_stop(5));
   start_pointer = start_pointer+find_stop;
   all_stops = find(redundancy_vector(start_pointer:len,3) <= ...
                    (interval_start(1)+max_dynrange2));
   stop_vector = redundancy_vector(all_stops+start_pointer-1,:);
   all_starts = find(stop_vector(:,1) >= interval_start(1));
end

% This section computes the sizes of the intervals and keeps only the
% largest intervals which are the dynamic ranges
len = length(intervals);
interval_sizes = [intervals(1:len-1,1)+1 intervals(1:len-1,4) ...
                  intervals(2:len,3)-1 intervals(2:len,4)];

% generating a shapefile for plotting all intervals
interval_vector = [interval_sizes(:,1) interval_sizes(:,3)
repmat(nan,size(interval_sizes,1),1)];
Generate_CircleArc_Shapefile(interval_vector,modli,'_Intervals');


interval_sizes(:,5) = interval_sizes(:,3)-interval_sizes(:,1)+1;
```

```
dynamic_range = interval_sizes(find(interval_sizes(:,5)== ...
                max(interval_sizes(:,5))),:);

% generating a shapefile for plotting all dynamic ranges
dynamic_range_vector = [dynamic_range(:,1) dynamic_range(:,3)
repmat(nan,size(dynamic_range,1),1)];
Generate_CircleArc_Shapefile(dynamic_range_vector,modli,'_DynamcRange');
```

3. Calculate_ArcView_Redundancies.m

```
function [redundancies] = Calculate_ArcView_Redundancies(cor,...
                                                         sortedBin,...
                                                         modli,...
                                                         CaseID)

%*************************************************************************
%* Calculate_ArcView_Redundancies Calculates all RNSN redundancies in the
%*     fundamental period for the all combinations and sub-cases of the
%*     input case and generates the text files ArcGIS uses to create
%*     shapefiles.
%*
%*     redundancies = Calculate_ArcView_Redundancies (cor, ...
%*                                                    sortedBin,modli,CaseID)
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 18AUG04
%*
%*     Called Functions: Generate_CircleArc_Shapefile
%*     Calling Functions: ArcViewOutput
%*
%*************************************************************************

global N Pf MAX_DYNRANGE MIN_DYNRANGE

redundancies = [];

% Finds all of the combinations that match the CaseID and stores them in a
% matrix called allBinaryNums which has 1's and 0's in each row that
% represents a particular of Type 1 and Type 0 redundancies.  Each row
% represents an individual combination of Type 1 and Type 0 redundancies
% for the particular case.
[rows,cols] = find(sortedBin(:,min(size(sortedBin))) == CaseID);
allBinaryNums = sortedBin(rows,1:N);
s = size(allBinaryNums,1);

% This loop goes through each combination and computes and labels the
% redundancies from 0 to Pf for every SUB-CHANNEL.
for ii = 1:s

    %form the string identifier for this particular case
    strID = [num2str(CaseID),num2str(ii)];

    % find all Type 1 and Type 0 redundancies
    f1 = find(allBinaryNums(ii,:)==1);
    f0 = find(allBinaryNums(ii,:)==0);

    % computing partial results for the length and COR of the redundancy
    m = prod(modli(f0));
    mm = prod(modli(f1));
```

```
    % redundancy length
    redundancy_len = 2*N*m;

    % compute all COR and labels for the 0th subchannel
    CORvector = [0:N*mm:Pf-1];
    numID = str2num([strID,'0']);
    tempCORvector = [CORvector' repmat(numID,length(CORvector),1)];

    % compute all COR and labels for all other SUB-CHANNELs
    for sub_chan = 1:N-1
        numID = str2num([strID,num2str(sub_chan)]);
        tempCORvector_shift = [CORvector'+cor(sub_chan+1) ...
                repmat(numID,length(CORvector),1)];
        tempCORvector = [tempCORvector; tempCORvector_shift];
    end

    % adjust COR so that they are all in a single Pf
    f = find(tempCORvector(:,1)>Pf);
    tempCORvector(f,1) = tempCORvector(f,1)-Pf;

    % compute start and stop endpoints in a vector of the form
    % [StartPoint StopPoint Length CaseLabel]
    redundancy_vector = [tempCORvector(:,1)-N*m tempCORvector(:,1)+N*m, ...
            tempCORvector(:,2)];

    % create shapefile
    Generate_CircleArc_Shapefile(redundancy_vector,modli,'_Case_');

    % continue only if the redundancy will affect the maximum dynamic range
    if redundancy_len <= MAX_DYNRANGE

        % Only keep COR located in 0 to Pf/2 since the rest are redundant
        f = find(tempCORvector(:,1)<=Pf/2);
        tempCORvector = tempCORvector(f,:);

        % compute start and stop endpoints in a vector of the form
        % [StartPoint COR StopPoint CaseLabel]
        startstop_vector = [tempCORvector(:,1)-N*m tempCORvector(:,1) ...
                tempCORvector(:,1)+N*m tempCORvector(:,2)];
        redundancies = [redundancies; startstop_vector];

        % compute the maximum dynamic range for this case and if it is
        % smaller than the previous maximum, replace with current max.
        case_max_dynrange = N*mm+redundancy_len-1;
        if MAX_DYNRANGE > case_max_dynrange
            MAX_DYNRANGE = case_max_dynrange;
        end
    end
end
```

4. Generate_RSNSCircle_Shapefile.m

```
function Generate_RSNSCircle_Shapefile(modli,fname)

%*************************************************************************
%* Generate_RSNSCircle_Shapefile  This function generates the RSNS circle
%*     plot text file that ArcGIS uses to generate a shapefile.  These files
%*     can get very long so only use for small channel numbers and moduli.
```

```
%*
%*     Usage: RSNS = Generate_RSNSCircle_Shapefile([m1 m2 ...],filename)
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 19AUG04
%*
%*     Called Functions: None
%*     Calling Functions: ArcViewOutput
%*
%************************************************************************

global RADIUS N Pf

% Set the width of each block in the RSNS circle
binwidth = (2*pi*RADIUS)/Pf;

% This loop creates the N individual channels in the RSNS Pf corresponding
% to each of the input moduli
for i = 1:N
    m = modli(i);

    % Make half on one period for one channel
    half_rsns = floor([0:(N*(m+1))-1]/N);

    % Make the other half and join into one period
    rsns = [half_rsns,fliplr(half_rsns(N+1:max(size(half_rsns))-N))];

    % If not the 1st channel, circularly shift the channel period
    % by the number of positions equal to one less than the channel number
    if i > 1
        rsns = [rsns(i:max(size(rsns))) rsns(1:i-1)];
    end

    % Replicate the channel period to a length equal to Pf
    rsnsPf(i,:) = repmat(rsns,1,ceil(Pf/max(size(rsns))));
end

% The following section generates the text file that ArcGIS uses to make a
% shapefile (RSNS circle plot)

% Check for a correct filename
if ischar(fname) ~= 1
    error('Not a valid filename')
end
fname = [fname '.txt'];
fid = fopen(fname,'wt');

% Provide labels for the columns (ArcGIS works in Longitude and Latitude)
fprintf(fid,'Lon,Lat,h_mod_N,h,RSNS\n');

% Create an isosceles trapezoid shape for each RSNS vector and associate
% corresponding RSNS vector information for the ArcGIS database
for rho = 0:Pf-1

    % ['0,0,0,0,New\n'] indicates a new shape
    fprintf(fid,['0,0,0,0,New\n']);

    % get the vector information
    chanels = num2str(rsnsPf(:,rho+1)');

    % compute the corner locations
    [corner(1,1),corner(1,2)] = pol2cart(rho*2*pi/Pf,RADIUS);
```

```
    [corner(2,1),corner(2,2)] = pol2cart((rho+1)*2*pi/Pf,RADIUS);
    [corner(3,1),corner(3,2)] = pol2cart((rho+1)*2*pi/Pf,RADIUS-binwidth);
    [corner(4,1),corner(4,2)] = pol2cart(rho*2*pi/Pf,RADIUS-binwidth);

    % write the corner locations and associated information to the file
    fprintf(fid,[num2str(corner(1,1),'%1.14f'),',',num2str(corner(1,2),...
            '%1.14f'),',',num2str(mod(rho,N)),',',num2str(rho),...
            ',"',chanels,'"\n']);
    fprintf(fid,[num2str(corner(2,1),'%1.14f'),',',num2str(corner(2,2),...
            '%1.14f'),',',num2str(mod(rho,N)),',',num2str(rho),...
            ',"',chanels,'"\n']);
    fprintf(fid,[num2str(corner(3,1),'%1.14f'),',',num2str(corner(3,2),...
            '%1.14f'),',',num2str(mod(rho,N)),',',num2str(rho),...
            ',"',chanels,'"\n']);
    fprintf(fid,[num2str(corner(4,1),'%1.14f'),',',num2str(corner(4,2),...
            '%1.14f'),',',num2str(mod(rho,N)),',',num2str(rho),...
            ',"',chanels,'"\n']);
    fprintf(fid,[num2str(corner(1,1),'%1.14f'),',',num2str(corner(1,2),...
            '%1.14f'),',',num2str(mod(rho,N)),',',num2str(rho),...
            ',"',chanels,'"\n']);
end

fclose(fid);
```

5. Generate_CircleArc_Shapefile.m

```
function [] = Generate_CircleArc_Shapefile(plot_circle_arcs,modli,arcType)

%*************************************************************************
%* Generate_CircleArc_Shapefile  This function generates the RSNS
%*     redundancy arcs text file that ArcGIS uses to generate a shapefile.
%*
%*     Author:        LCDR Brian Luke
%*     Last Modified: 19AUG04
%*
%*     Called Functions: None
%*     Calling Functions: Calculate_ArcView_Redundancies,
%*                        ArcViewOutput
%*
%*************************************************************************

global RADIUS N Pf

% Set the width of each arc in the RSNS circle
binwidth = (2*pi*RADIUS)/Pf;

% find each case
CaseLabels = unique(plot_circle_arcs(:,3));

% generate a seperate shapfile for each redundancy case
for ii = 1:length(CaseLabels)
    f = find(plot_circle_arcs(:,3)==CaseLabels(ii));
    circle_arc = plot_circle_arcs(f,:);

    % generate the filename, open the file, and label the data fields
    fname = [num2str(N),'Channel'];
    for i = 1:N
        fname = [fname,'_',num2str(modli(i))];
    end
```

```
    if isnan(CaseLabels(ii))==1
        fname = [fname, arcType, '.txt'];
    else
        fname = [fname, arcType, num2str(CaseLabels(ii)), '.txt'];
    end
    fid = fopen(fname,'wt');
    fprintf(fid,'Lon,Lat,NotUsed,length,Case\n');

    % arc lengths
    circle_arc_length = circle_arc(:,2)-circle_arc(:,1)+1;

    % Make the text for the Shapefile
    for i = 1:size(circle_arc,1)

        clear corner
        fprintf(fid,['0,0,0,0,New\n']);

        endln = repmat('\n',circle_arc_length(i)+1,1);
        notused = repmat(',0,',circle_arc_length(i)+1,1);
        len = repmat(num2str(circle_arc_length(i)), ...
                    circle_arc_length(i)+1,1);
        case_string = repmat([',Case ',num2str(CaseLabels(ii))],...
                            circle_arc_length(i)+1,1);

        [corner(1:circle_arc_length(i)+1,1),...
         corner(1:circle_arc_length(i)+1,2)] = pol2cart((...
                                               [circle_arc(i,1):...
                                               circle_arc(i,2)+1]')...
                                               *2*pi/Pf,RADIUS);

        fprintf(fid,[num2str(corner(1:circle_arc_length(i)+1,1),...
                    '%+1.14f,'),num2str(corner...
                    (1:circle_arc_length(i)+1,2),'%+1.14f'),...
                    notused,len,case_string,endln]');

        [corner(circle_arc_length(i)+2:...
                circle_arc_length(i)*2+2,1),...
                corner(circle_arc_length(i)+2:...
                circle_arc_length(i)*2+2,2)] = ...
                pol2cart([circle_arc(i,2)+1:-1:circle_arc(i,1)]')...
                *2*pi/Pf,RADIUS-binwidth*(mod(CaseLabels(ii),N)+1));

        fprintf(fid,[num2str(corner(circle_arc_length(i)+2:...
                    circle_arc_length(i)*2+2,1),'%+1.14f,'),...
                    num2str(corner(circle_arc_length(i)+2:...
                    circle_arc_length(i)*2+2,2),'%+1.14f'),...
                    notused,len,case_string,endln]');

        if isempty(corner) == 0
            fprintf(fid,[num2str(corner(1,1),'%+1.14f,'),...
                        num2str(corner(1,2),'%+1.14f,'),...
                        num2str(0),',',num2str(circle_arc_length(i)),...
                        ',Case ',num2str(CaseLabels(ii)),'\n']);
        end
    end
    fclose(fid);
end
```

6. ArcGIS shapefile generation visual basic code

```
Sub MultipleFiles()

    '*************************************************************
    ' MultipleFiles()  This is the main function that reads in a
    '     set of text files created by MATLAB that represent
    '     RSNS redundancies.
    '
    '     Modify the parameters strShapePath, N, and intArrayModuli
    '     below before running this function.
    '
    '     Author: LCDR Brian Luke
    '     Last Modified: 20AUG04
    '
    '     This code was created using examples of shapefile
    '     creation code from the ESRI ArcGIS website
    '*************************************************************

    ' loop indexes
    Dim i As Integer
    Dim ii As Integer
    Dim iii As Integer

    ' Number of channels and moduli
    Dim N As Integer
    Dim intArrayModuli As Variant

    ' Strings for filenames
    Dim strCaseFiles As String
    Dim strDynRangeFiles As String
    Dim strArrayFiles() As String
    Dim strShapePath As String
    Dim strPrefix As String
    Dim strShapeName As String

    '*****************************************************************
    ' Change these parameters to use program
    '
    ' strShapePath is the path to the text files and where the shape
    ' files will be placed after creation
    '
    ' N is the number of channels
    '
    ' intArrayModuli are the individual moduli
    '*****************************************************************

    strShapePath = "C:\this_path\this_directory\"
    N = 3
    intArrayModuli = Array(3, 4, 5)

    '*****************************************************************

    ' form filenames to match MATLAB filenames
    strPrefix = CStr(N) + "Channel_"
    For i = 1 To N
        strPrefix = strPrefix + CStr(intArrayModuli(i - 1)) + "_"
    Next

    strCaseFiles = "CirclePlot DynamicRange Intervals"
```

```
Dim combinations As Integer

For i = 1 To N
    combinations = Factorial(N) / (Factorial(i) * (Factorial(N - i)))
    For ii = 1 To combinations
        For iii = 0 To N - 1
            strCaseFiles = strCaseFiles + _
                           " Case_" + _
                           CStr(i) + _
                           CStr(ii) + _
                           CStr(iii)
        Next
    Next
Next

' split filenames into an array and loop through each file
strArrayFiles = Split(strCaseFiles)

Dim strFilename
Dim strTempFilename() As String
Dim strDynRge As String
Dim intDynExists As Integer

For Each strFilename In strArrayFiles

' shapefile name that appears on the ArcGIS display
strShapeName = strFilename

'form complete path to MATLAB text file
Dim strInputTextFile As String
strInputTextFile = strShapePath + strPrefix + strFilename + ".txt"

'create the shapefiles
Dim pFeatureClass As IFeatureClass
Set pFeatureClass = CreatePolygonShapefile(strShapePath, strShapeName)

'open text file
Close #1
Open strInputTextFile For Input As #1

Dim dblLon As Double
Dim dblLat As Double
Dim pPoint As IPoint
Dim pPointCollection As IPointCollection
Dim int_Field1 As Integer
Dim int_Field2 As Integer
Dim str_Field3 As String
Dim int_Field1Value As Integer
Dim int_Field2Value As Integer
Dim str_Field3Value As String

'Get the header
Dim strHeader As String
Line Input #1, strHeader

' Loop through the file forming each polygon
Dim lngCounter As Long
lngCounter = 0
Do While Not (EOF(1))
    'input the data row and create the new polygon point
    Input #1, dblLon, dblLat, int_Field1, int_Field2, str_Field3
    lngCounter = lngCounter + 1
```

```
        Set pPoint = New Point
        pPoint.PutCoords dblLon, dblLat

        If str_Field3 = "New" Then

            If lngCounter = 1 Then
                'first polygon, don't write previous
            Else
                'write previous polygon
                Call AddPointCollection2FeatureClassPOL(pFeatureClass, _
                                                        pPointCollection, _
                                                        int_Field1Value, _
                                                        int_Field2Value, _
                                                        str_Field3Value)

            End If

            'start the new pointcollection
            Set pPointCollection = New Polygon

        ElseIf str_Field3 <> "" Then

            'add the data fields and add the point to the polygon
            int_Field1Value = int_Field1
            int_Field2Value = int_Field2
            str_Field3Value = str_Field3
            pPointCollection.AddPoint pPoint

        End If

    Loop
    Close #1

    'write last polygon
    Call AddPointCollection2FeatureClassPOL(pFeatureClass, _
                                            pPointCollection, _
                                            int_Field1Value, _
                                            int_Field2Value, _
                                            str_Field3Value)

    'Add the shape file to ArcMap, label it, and make it 50% transparent
    Dim pFLayer As IFeatureLayer
    Dim pFLayerEffects As ILayerEffects
    Set pFLayer = New FeatureLayer
    Set pFLayer.FeatureClass = pFeatureClass
    pFLayer.name = pFeatureClass.AliasName
    Set pFLayerEffects = pFLayer
    pFLayerEffects.Transparency = 50

    'zoom the map to the new layer
    Dim pMxDoc As IMxDocument
    Set pMxDoc = ThisDocument
    pMxDoc.FocusMap.AddLayer pFLayer

    'refresh map
    pMxDoc.ActiveView.Refresh
    pMxDoc.UpdateContents

Next

MsgBox "Done..."
```

```
End Sub


Public Function CreatePolygonShapefile(strFolder As String, _
                                       strShapeName As String) _
                                       As IFeatureClass

    '*************************************************************
    ' CreatePolygonShapefile(strFolder,strShapeName)
    '     This function creates a polygon shapefile in the folder
    '     and with the name specified by the input.  This function
    '     also adds the numeric and string fields that will be in
    '     the database table associated with the shapes in the
    '     layer.
    '*************************************************************

    Const strShapeFieldName As String = "Shape"

    ' Open the folder to contain the shapefile as a workspace
    Dim pFWS As IFeatureWorkspace
    Dim pWorkspaceFactory As IWorkspaceFactory
    Set pWorkspaceFactory = New ShapefileWorkspaceFactory
    Set pFWS = pWorkspaceFactory.OpenFromFile(strFolder, 0)

    ' Set up a simple fields collection
    Dim pFields As IFields
    Dim pFieldsEdit As IFieldsEdit
    Set pFields = New esriCore.Fields
    Set pFieldsEdit = pFields

    Dim pField As IField
    Dim pFieldEdit As IFieldEdit

    ' Make the polygon shape field
    Set pField = New esriCore.Field
    Set pFieldEdit = pField
    pFieldEdit.name = strShapeFieldName
    pFieldEdit.Type = esriFieldTypeGeometry

    Dim pGeomDef As IGeometryDef
    Dim pGeomDefEdit As IGeometryDefEdit
    Set pGeomDef = New GeometryDef
    Set pGeomDefEdit = pGeomDef
    With pGeomDefEdit
        .GeometryType = esriGeometryPolygon
        Set .SpatialReference = New UnknownCoordinateSystem
    End With
    Set pFieldEdit.GeometryDef = pGeomDef
    pFieldsEdit.AddField pField

    ' Add Field 1, integer, h (mod N)
    Set pField = New esriCore.Field
    Set pFieldEdit = pField
    With pFieldEdit
      .name = "hModN"
      .Type = esriFieldTypeInteger
    End With
    pFieldsEdit.AddField pField

    ' Add Field 2, integer, vector position h
    Set pField = New esriCore.Field
    Set pFieldEdit = pField
```

```
    With pFieldEdit
      .name = "h"
      .Type = esriFieldTypeInteger
    End With
    pFieldsEdit.AddField pField

    ' Add Field 3, string, shape description (i.e., case label)
    Set pField = New esriCore.Field
    Set pFieldEdit = pField
    With pFieldEdit
      .name = "Case"
      .Type = esriFieldTypeString
      .Length = 30
    End With
    pFieldsEdit.AddField pField

    ' Create the shapefile
    Dim boolIsDeleted As Boolean
    boolIsDeleted = DeleteShapeFile(strFolder, strShapeName)

    'Make shapefile as featureclass
    Set CreatePolygonShapefile = pFWS.CreateFeatureClass(strShapeName, _
                                                         pFields, _
                                                         Nothing, _
                                                         Nothing, _
                                                         esriFTSimple, _
                                                         strShapeFieldName, _
                                                         "")

End Function


Public Function DeleteShapeFile(strFolder As String, _
                                strShapeName As String) _
                                As Boolean

    '*************************************************************
    ' DeleteShapeFile(strFolder,strShapeName)
    '     This function deletes a polygon shapefile in the folder
    '     and with the name specified by the input.  This is used
    '     when a file is to be overwritten.
    '*************************************************************

    On Error GoTo EH

    Dim pDS As IDataset
    Dim DeleteFile As Boolean

    ' try shapefile and delete if it exists
    Dim pFClass As IFeatureClass
    Set pFClass = OpenShapeFile(strFolder, strShapeName)
    If (Not pFClass Is Nothing) Then
        Set pDS = pFClass
        pDS.Delete
        DeleteFile = True
        Exit Function
    End If

EH:
    DeleteFile = False

End Function
```

```
Public Function OpenShapeFile(dir As String, _
                              name As String) _
                              As IFeatureClass

    '*************************************************************
    ' OpenShapeFile(dir,name)
    '     This function opens a shapefile for adding data.
    '*************************************************************

    Dim pWSFact As IWorkspaceFactory
    Dim connectionProperties As IPropertySet
    Dim pShapeWS As IFeatureWorkspace
    Dim isShapeWS As Boolean

    Set OpenShapeFile = Nothing

    Set pWSFact = New ShapefileWorkspaceFactory
    isShapeWS = pWSFact.IsWorkspace(dir)
    If (isShapeWS) Then
        On Error GoTo ErrHandler
        Set connectionProperties = New PropertySet
        connectionProperties.SetProperty "DATABASE", dir
        Set pShapeWS = pWSFact.Open(connectionProperties, 0)
        Dim pFClass As IFeatureClass
        Set pFClass = pShapeWS.OpenFeatureClass(name)
        Set OpenShapeFile = pFClass
    End If

ErrHandler:

End Function


Sub AddPointCollection2FeatureClassPOL(pFeatureClass As IFeatureClass, _
                                       pPointCollection As IPointCollection, _
                                       int_Field1 As Integer, _
                                       int_Field2 As Integer, _
                                       str_Field3 As String)

    '*************************************************************
    ' AddPointCollection2FeatureClassPOL(pFeatureClass,pPointCollection,
    '                                    int_Field1,int_Field2,str_Field3)
    '     This function adds a shape to a layer and adds the associated
    '     data to the database.
    '*************************************************************

    Dim pFeature As IFeature
    Set pFeature = pFeatureClass.CreateFeature
    Set pFeature.Shape = pPointCollection
        pFeature.Value(pFeature.Fields.FindField("hModN")) = int_Field1
        pFeature.Value(pFeature.Fields.FindField("h")) = int_Field2
        pFeature.Value(pFeature.Fields.FindField("Case")) = str_Field3
    pFeature.Store

End Sub


Function Factorial(N)

    '*************************************************************
    ' Factorial(N)
```

```
    '     This function computes the factorial of a number (N!)
    '*************************************************************

    If N <= 1 Then
        Factorial = 1
    Else
        Factorial = Factorial(N - 1) * N
    End If
End Function
```

7. Sample Output of ArcGIS Shapefile Generation Visual Basic Code

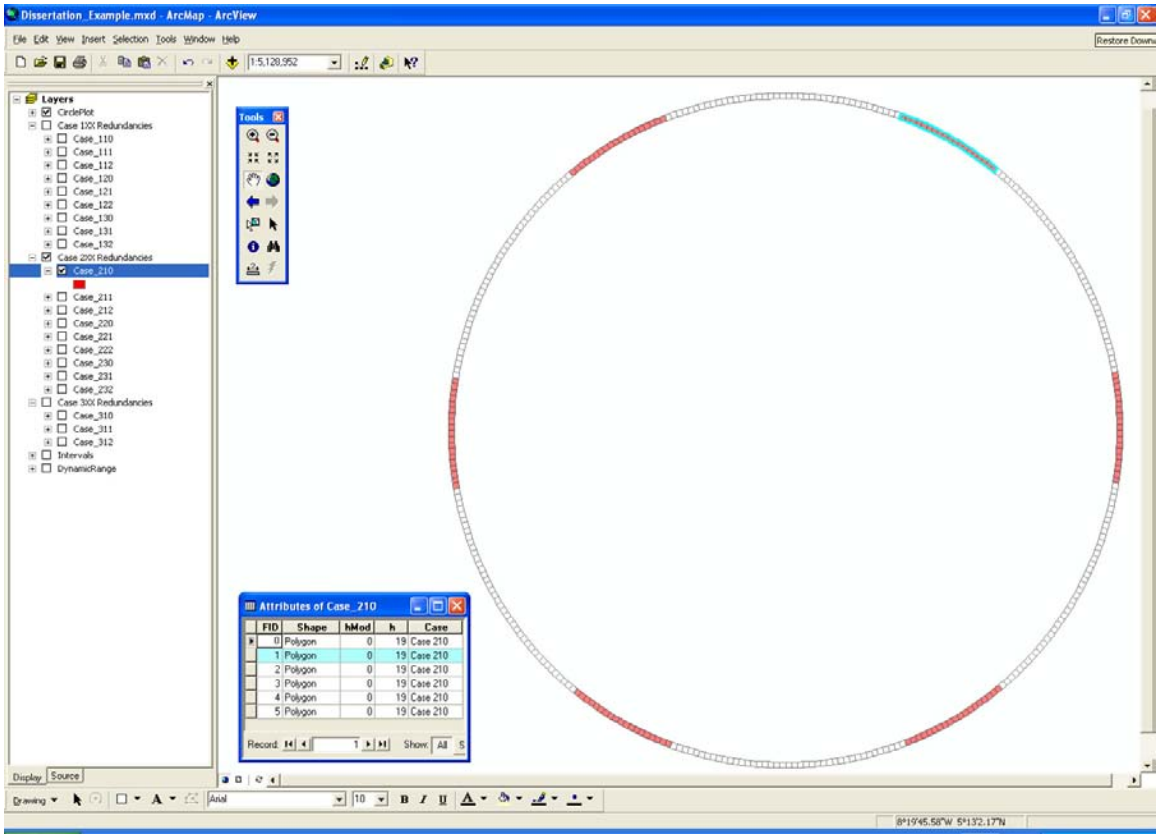

Figure C.3 Screen shot of the ArcGIS map showing the RSNS circle plot, a Case 210 redundancy, and associated attribute table. A single redundancy is selected on the map and in the linked table.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Office of Naval Research, "A Future Naval Capability: Missile Defense." [http://www.onr.navy.mil/media/extra/fncs_fact_sheets/missile_defense.pdf]. Last accessed Sep. 2004.

[2] Office of Naval Research, "A Future Naval Capability: Platform Protection." [http://www.onr.navy.mil/media/extra/fncs_fact_sheets/platform_protect.pdf]. Last accessed Sep. 2004.

[3] N. J. Porter, R. J. A. Tough, and K. D. Ward, *ISAR Imaging of Maritime Targets: Theory and Simulation*, Defence Research Agency Malvern report 4577 Sep. 1992.

[4] R. Madden, *An Introduction to the Functionality of the ISAR Radar,* Naval Research Laboratory Radar Division document NRL/MR/5313-01-8533, Feb. 2001.

[5] P. E. Pace and G. D. Burton, "Antiship Cruise Missiles: Technology, Simulation and Self-Defense," *Journal of Electronic Defense*, vol. 4, pp. 11-15, Nov. 1998.

[6] C. J. Condley, "Some System Considerations for Electronic Countermeasures to Synthetic Aperture Radar," *Proceedings of the IEE Colloquium on Electronic Warfare Systems*, pp. 8/1-8/7, 14 Jan. 1991.

[7] S. L. Fearnley and E. P. Meakin, "EW Against Anti-Ship Missiles," *Proceedings of the IEE Colloquium on Signal Processing Techniques for Electronic Warfare*, pp. 7/1-7/4, 31 Jan. 1992.

[8] D. J. Fouts, P. E. Pace, C. Karow, and S. R. T. Ekestorm, "A Single-Chip False Target Radar Image Generator for Countering Wideband Imaging Radars," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 751-759, Jun. 2002.

[9] P. E. Pace, D. J. Fouts, C. Karow, and S. R. T. Ekestorm, "Digital False-Target Image Synthesizer for Countering ISAR," *IEE Proceedings on Radar Sonar Navigation*, vol. 149, pp. 248-257, Oct. 2002.

[10] G. G. E. Gielen and R. A. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits," *Proceedings of the IEEE*, vol. 88, pp. 1825-1854, Dec. 2000.

[11] K. Kundert, H. Chang, D. Jefferies, G. Lamant, E. Malavasi, and F. Sendig, "Design of Mixed-Signal Systems-on-a-Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1561-1571, Dec. 2000.

[12] A. Finn, K. Brown, and T. Lindsay, "Miniature UAVs & Future Electronic Warfare," presented at the Land Warfare Conference, Brisbane, Australia, 22 Oct. 2002. [http://www.aerosonde.com/downloads/Aerosonde_DSTO_EW.pdf]. Last accessed Sep. 2004.

[13] D. Ledger, "Electronic Warfare Capabilities of Mini UAVs," presented at the 2002 Electronic Warfare Conference, Kuala Lumpur, 2002. [http://www.aerosonde.com/downloads/electronic_warfare_ledger.doc]. Last accessed Sep. 2004.

[14] D. L. Harame, D. C. Ahlgren, D. D. Coolbaugh, J. S. Dunn, G. G. Freeman, J. D. Gillis, R. A. Groves, G. N. Hendersen, R. A. Johnson, A. J. Joseph, S. Subbanna, A. M. Victor, K. M. Watson, C. S. Webster, and P. J. Zampardi, "Current Status and Future Trends of SiGe BiCMOS Technology," *IEEE Transactions on Electron Devices*, vol. 48, pp. 2575-2594, Nov. 2001.

[15] E. Liu, C. Wong, Q. Shami, S. Mohapatra, R. Landy, P. Sheldon, and G. Woodword, "Complete Mixed-Signal Building Blocks for Single-Chip GSM Baseband Processing," *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference*, pp. 101-104, 11 May 1998.

[16] D. Leenaerts, G. Gielen, and R. Rutenbar, "CAD Solutions and Outstanding Challenges for Mixed-Signal and RF IC Design," P*roceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 270-277, 4 Nov. 2001.

[17] R.C. Altmeyer, "Design, Implementation, and Testing of a VLSI High Performance ASIC for Extracting the Phase of a Complex Signal," Master's thesis, Naval Postgraduate School, Monterey, California, Sep. 2002.

[18] P. E. Pace, D. J. Fouts, C. Karow, and S. Ekestorm, "An All-Digital Image Synthesizer for Countering High-Resolution Imaging Radars," Naval Postgraduate School Technical Report, NPS-EC-00-005, Feb. 2000.

[19] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press: New York, 2000.

[20] C. Huang, J. Wang, and Y. Huang, "A High-speed CMOS Incrementer/Dec-rementer," *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 88-91, May 2001.

[21] J. Yoo, G. Gopalakrishnan, and K. Smith, "Timing Constraints for High-Speed Counterflow-Clocked Pipelining," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 2, pp. 167-173, Jun. 1999.

[22] Y. Elboim, A. Kolodny, and R. Ginosar, "A Clock-Tuning Circuit for System-on-Chip," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 4, pp. 616-626, Aug. 2003.

[23] S. H. Embabi and D. E. Brueske, "Clock Synchronization for WSI Systems," *Proceedings of the IEEE International Conference on Wafer Scale Integration*, pp. 228-234, Jan. 1994.

[24] D. E. Brueske and S. H. Embabi, "A Dynamic Clock Synchronization Technique for Large Systems," *IEEE Transactions on Components, Packaging, and Manufacturing Technology*, vol. 17, no. 3, pp. 350-361, Aug. 1994.

[25] H. Lee, H. Q. Nguyen, and D. W. Potter, "Design Self-Synchronized Clock Distribution Networks in an SOC ASIC Using DLL with Remote Clock Feedback," *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference*, pp. 248-252, Sep. 2000.

[26] The MOSIS Service, AMI Semiconductor 1.5 micron ABN process. [http://www.mosis.org/products/fab/vendors/amis/abn/]. Last accessed Oct. 2004.

[27] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, Addison-Wesley: New York, 1994.

[28] P. E. Pace, *Advanced Techniques for Digital Receivers*, Artech House: Norwood, Massachusetts, 2000.

[29] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design*, Oxford University Press: New York, 2002.

[30] P. E. Pace, D. Styer, and I. A. Akin, "A Folding ADC Preprocessing Architecture Employing a Robust Symmetrical Number System with Gray-Code Properties," *IEEE Trans. Circuits Syst. II*, vol. 47, pp. 462-467, May 2000.

[31] S. Subbanna, G. Freeman, D. Ahlgren, D. Greenberg, D. Harame, J. Dunn, D. Herman, B. Meyerson, Y. Greshishchev, P. Schvan, D. Thornberry, G. Sakamoto, and R. Tayrani, "Integration and Design Issues in Combining Very-High-Speed Silicon-Germanium Bipolar Transistors and ULSI CMOS for System-on-a-Chip Applications," *IEEE International Electron Devices Meeting*, pp. 845-848, Sep. 1999.

[32] IBM technical report G522-0353-00, *BlueLogic BiCMOS 5HP Technology: SiGe BiCMOS process for high performance*, IBM Microelectronics Division, New York, Jun. 1998.

[33] D. Styer and P. E. Pace, "Two Channel RSNS Dynamic Range," *IEEE Trans. Circuits Syst. I*, vol. 49, pp. 395-397, Mar. 2002.

[34] P. E. Pace, D. Wickersham, D. C. Jenn, and N. S. York, *"*High-Resolution Phase Sampled Interferometry Using Symmetrical Number Systems," *IEEE Transactions on Antennas and Propagation*, vol. 49, pp. 1411–1423, Oct. 2001.

[35] P. E. Pace, D. C. Jenn, and J. P. Powers, "Symmetrical Number Systems: Theory and Applications," *Transworld Research Network Journal*, pp. 91-121, Jan. 2002.

[36] P. E. Pace, R. E. Leino, and D. Styer, "Use of the Symmetrical Number System in Resolving Single-Frequency Undersampling Ambiguities," *IEEE Trans. on Signal Processing*, vol. 45, pp. 1153–1160, May 1997.

[37] S. Andraos and H. Ahmad, "A New Efficient Memoryless Residue to Binary Converter," *IEEE Trans. Circuits and Systems*, vol. 35, pp. 1441-1444, Nov. 1988.

[38] ESRI, "What is GIS?" ESRI PDF presentation. [http://www.gis.com/whatisgis/whatisgis.pdf.]. Last accessed Sep. 2004.

[39] ESRI, "Geography Matters," ESRI white paper, ESRI, Redlands, CA, Sep. 2002. [http://www.gis.com/whatisgis/geographymatters.pdf.]. Last accessed Sep. 2004.

[40] ESRI, "What is GIS?" ESRI tutorial document. [http://www.gis.com/whatisgis/index.html.]. Last accessed Sep. 2004.

[41] B. L. Luke, P. E. Pace, and D. Styer, "Three Channel RSNS Dynamic Range," paper submitted to the *IEEE Trans. Circuits Syst. I*.

[42] P.E. Pace and D. Styer, "An Optimum SNS-to-Binary Conversion Algorithm and Pipelined Field-Programmable Logic Design," *IEEE Trans. Circuits Syst. I*, vol. 47, pp. 736-745, Aug. 2000.

[43] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall: Englewood Cliffs, New Jersey, 1983.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

4. Professor Douglas Fouts (EC/Fs)
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

5. Professor Phillip Pace (EC/Pc)
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

6. Professor Herschel Loomis (EC/Lm)
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

7. Professor Charles Therrien (EC/Ti)
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

8. Professor Cynthia Irvine (CS/Ic)
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

9. Commander Naval Security Group, Code N6
   Fort Meade, MD

10. Commanding Officer, Naval Information Warfare Activity
    Fort Meade, MD

11. Dr. Peter Craig, Office of Naval Research, Code 313
Arlington, VA

12. Dr. Chip Grounds, Office of Naval Research, Code 313
Arlington, VA

13. Mr. Mike Monsma, Office of Naval Research, Code 313
Arlington, VA

14. Mr. Jim Talley, Office of Naval Research, Code 313
Arlington, VA

15. Mr. Greg Hrin, Naval Research Laboratory, Code 5740
Washington, DC

16. Dr. Ted Roberts, Naval Research Laboratory, Code 5720
Washington, DC

17. Mr. Alfred DiMatessa, Naval Research Laboratory, Code 5700
Washington, DC

18. Dr. Frank Klemm, Naval Research Laboratory, Code 5700
Washington, DC

19. Commissioner Dennis Luke and Mrs. Joanne Luke
Bend, OR

20. Mr. Paul Young and Mrs. Pat Young
Pensacola, FL

21. LCDR Brian Luke
Naval Information Warfare Activity
Fort Meade, MD

DEPARTMENT OF THE NAVY
NAVAL POSTGRADUATE SCHOOL
DUDLEY KNOX LIBRARY
411 DYER ROAD, ROOM 110
MONTEREY, CALIFORNIA 93943-5101

D300
NPS (130)
26 Jan 11

From: Dudley Knox Library, Naval Postgraduate School (13)
To: Defense Technical Information Center (DTIC-OQ)

Subj: CHANGE OF DISTRIBUTION STATEMENT FOR ADB310230

1. Request a distribution statement change for:

ADB310230: Luke, Brian L. *Architecture of an Integrated Microelectronic Warfare System on a Chip and Design of Key Components*. Monterey, CA: Naval Postgraduate School. Department of Electrical and Computer Engineering, December 2004. UNCLASSIFIED [Distribution authorized to U.S. Government Agencies only; premature dissemination; December 2004]

2. Upon consultation with NPS faculty, the School has determined that this thesis may be released to the public and that its distribution is unlimited, effective January 25, 2011.

3. POC for this request is George Goncalves, Librarian, Restricted Resources and Services, 831-656-2061, DSN 756-2061 (gmgoncal@nps.edu).

Valerie A. Moulé

VALERIE A. MOULÉ
Associate University Librarian