1997

Design and implementation of high-radix arithmetic systems based on the SDNR/RNS data representation

Paul Whyte
Edith Cowan University

Edith Cowan University

Copyright Warning

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

EDITH COWAN UNIVERSITY
FACULTY OF SCIENCE, TECHNOLOGY AND ENGINEERING
SCHOOL OF MATHEMATICS, INFORMATION TECHNOLOGY AND ENGINEERING
DEPARTMENT OF COMPUTER AND COMMUNICATION ENGINEERING

ENS4241 Engineering Project 2

Project report

Design and implementation of high-radix arithmetic systems based on the SDNR/RNS data representation

Student: Paul Whyte (0930227)
Project coordinator: Associate Professor Wojciech Kuczborski
Project examiners: Associate Professor Wojciech Kuczborski
Dr Stefan Lachowicz
Date: Thursday, 16 January 1997

***

Acknowledgments

I would like to express my sincere gratitude to my supervisor for this project, Associate Professor Wojciech Kuczborski, for his advice and guidance. I am also thankful to my family for their support during both this project and my undergraduate university career.

Paul Whyte

Abstract

This project involved the design and implementation of high-radix arithmetic systems based on the hybrid SDNR/RNS data representation. Some real-time applications require a real-time arithmetic system. An SDNR/RNS arithmetic system provides parallel, real-time processing.

The advantages and disadvantages of high-radix SDNR/RNS arithmetic, and the feasibility of implementing SDNR/RNS arithmetic systems in CMOS VLSI technology, were investigated in this project. A common methodological model, which included the stages of analysis, design, implementation, testing, and simulation, was followed.

The combination of the SDNR and RNS transforms potential complex logic networks into simpler logic blocks. It was found that when constructing a SDNR/RNS adder, factors such as the radix, digit set, and moduli must be taken into account.

There are many avenues still to explore. For example, implementing other arithmetic systems in the same CMOS VLSI technology used in this project and comparing them to equivalent SDNR/RNS systems would provide a set of benchmarks. These benchmarks would be useful in addressing issues relating to relative performance.

I certify that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief in does not contain any material previously published or written by another person except where due reference is made in the text.

Signature

Date 16.1.97

Table of contents

Terminologies

Table 1: Arithmetic symbols.

Symbol	Meaning	Alias
$\overline{X}$	Negative SDNR digit -X.	
δ	Diminished cardinality.	
ω	Offset.	
$\in$	Element of.	
{...}	Set.	
<...>	RNS number.	
a	Maximum digit in SDNR digit set.	
b	Base.	r
CEILING (number, significance)	Returns number rounded up, away from zero, to the nearest multiple of significance. number is the value to be rounded. significance is the multiple to which number should be rounded. Default significance = 1.	
FLOOR (number, significance)	Rounds number down, toward zero, to the nearest multiple of significance. number is the numeric value to be rounded. significance is the multiple to which number should be rounded. Default significance = 1.	
INT (number)	Rounds a number down to the nearest integer. number is the real number to be rounded down to an integer.	
$\log_b n$	Finds the logarithm of the number n with respect to the base b.	
p	Element in an RNS moduli set.	
P (...)	Probability	
r	Radix.	b
SIGN (number)	Returns the sign of number.	
t	Threshold value.	

Table 2: Abbreviations.

Abbreviation	Meaning
CAD	Computer Aided Design.
CMOS	Complementary Metal Oxide Semiconductor.
DFT	Discrete Fourier Transform.
DRC	Diminished Radix Complement code.
FFT	Fast Fourier Transform.
FPGA	Field Programmable Gate Array.
GaAs	Gallium Arsenide.
I/O	Input/Output.
LSB	Least Significant Bit.
LSD	Least Significant Digit.
MSB	Most Significant Bit.
MSD	Most Significant Digit.
PE	Processing Element.
RC	Radix Complement code.
RNS	Residue Number System.
SDNR	Signed Digit Number Representation.
VLSI	Very Large Scale Integration.

1. Introduction

This project was concerned with the design and implementation of high-radix arithmetic systems based on the hybrid SDNR/RNS data representation. An arithmetic system is an entity which can perform one or more of the core mathematical operations which are addition, subtraction, multiplication, and division. Furthermore, an arithmetic system may also provide extra functionality with operations such as sign and overflow detection, and magnitude comparisons. The SDNR/RNS data representation allows high-radix arithmetic to be executed in a parallel, real-time fashion.

To distinguish between research already performed in the field of arithmetic systems and activities undertaken as a part of this project, this report has been divided up into two main sections, which are described in Table 3.

Table 3: Project report outline.

Part	Chapter	Description
-	1	Introduction.
	2	Project definition.
1	3	Background theory.
2	4	Analysis.
	5	Design.
	6	Testing.
	7	Implementation.
	8	Simulations.
	9	Conclusion.

The background theory chapter identifies the problems of current arithmetic systems used in computer systems. The chapter goes on to describe why, for a select group of applications, nonconventional data representations are needed, in particular, the SDNR, RNS, and SDNR/RNS number systems. Following this, general computer architectures and VLSI technologies are discussed.

The analysis chapter focuses on identifying the main characteristics of the SDNR/RNS number system. As a part of this chapter, a set of recommendations detailing how to choose an optimal SDNR/RNS configuration are presented.

Chapter five focuses on design. For this project, several components of the SDNR/RNS arithmetic system were designed, including an adder, sign detector, and conversion circuitry.

Chapter seven includes a discussion on the issues associated with the VLSI implementation. From the modules designed, the adder was the only arithmetic component to be implemented. Suggestions given during this chapter detail how the adder could have been implemented more effectively.

The testing chapter includes a description on how the adder was tested. The simulation chapter states project results based on simulations performed on the adder.

2. Project definition

2.1 Aim

The aim of this project was to design and simulate a high-radix arithmetic system based on SDNR/RNS data representation. The main objectives relating to this aim were:

1. Investigating the advantages and disadvantages of high-radix SDNR/RNS arithmetic over other conventional and non-conventional schemes.
2. Determining the feasibility of implementing the SDNR/RNS arithmetic system in CMOS VLSI technology.

2.2 Scope

The scope of this project involved conducting an analysis of the SDNR/RNS number system, and designing SDNR/RNS arithmetic systems and implementing them using software available in the VLSI Research Laboratory. Initially, the scope involved implementing and simulating several arithmetic circuits, including an adder, sign detector, comparator, and number system converters. However, only one of the circuits, the digit adder, was eventually realised. The main reason for not completely fulfilling the initial scope statement was due to time constraints.

2.3 Strategy

This report includes the analysis, design, implementation, testing, and simulation of an SDNR/RNS arithmetic system. Figure 1 shows a diagrammatic guide which was not only followed throughout the duration of the project, but canvases what is ahead in future chapters.

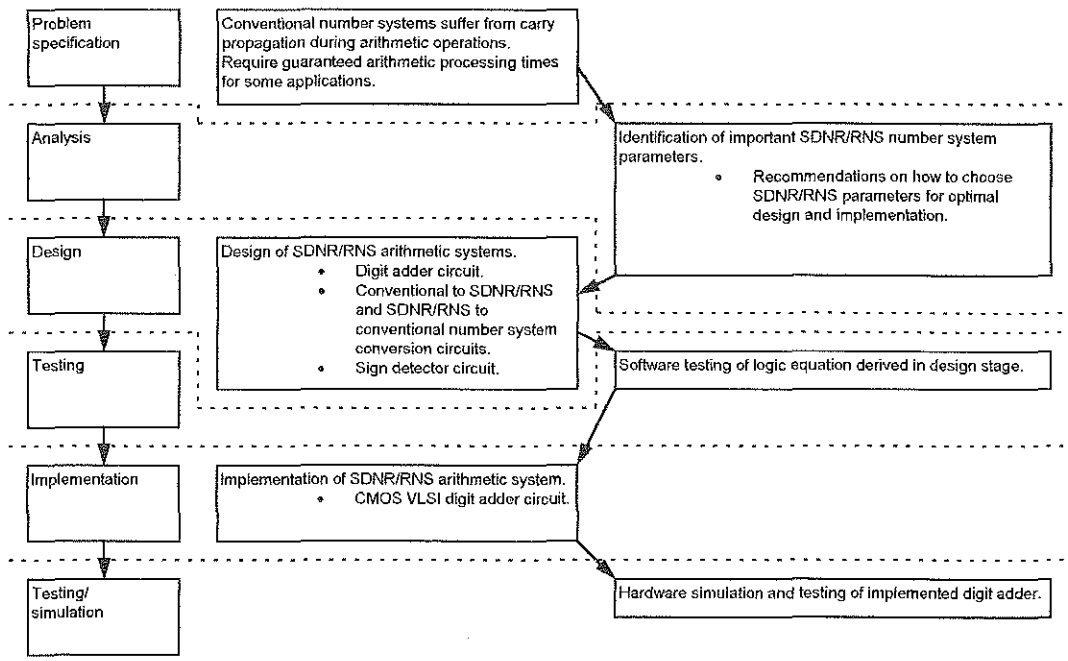

Figure 1: Outline of project.

3. Background theory

In this chapter, several topics will be explored. First, general computer arithmetic is discussed. From here, the advantages and disadvantages of using conventional notation, that is, binary, in performing digital arithmetic will be identified. Next, nonconventional number systems are introduced, namely the SDNR and RNS. Both of these number systems have the ability to overcome the limitations of conventional number arithmetic for certain applications. A description of a hybrid SDNR/RNS follows, which includes an explanation as to why such a hybrid scheme is needed.

The final section of this chapter concerns itself with design and implementation issues. For this project, a SDNR/RNS digit adder was designed and implemented using CMOS technology. Therefore, as a part of this final section, CMOS technologies, as well as general computer architectures and clocking schemes are discussed.

3.1 Computer arithmetic

A digital computer uses the binary number system to perform specified arithmetic. Pedler (1993) defines a number as an abstract idea represented by a word and a symbol. Particular sets of numbers, among others, are integers and real numbers. Pedler also describes a numeral as a symbol for a number. Thus, a numeration system is an orderly system for representing numbers as numerals.

Waser and Flynn (1982) point out that the main problem in computer arithmetic is the mapping from the human infinite number system to the finite representational capability of the machine. Garner (cited in Waser and Flynn, 1982) has shown that the most important characteristic of machine number systems is finitude. Nearly all other considerations are a direct consequence of the finitude. That is, overflows, underflows, scaling, and compliment coding are consequences of this finitude. Overflow, for example, is simply an unsuccessful attempt to map from the infinite to the finite number system.

3.1.1 Modular arithmetic

The common solution to this problem is the use of modular arithmetic (Waser and Flynn, 1982). This allows every integer from the infinite number set to be assigned to one unique representation in a finite system.

Waser and Flynn (1982) assert that in modular arithmetic, the property of congruence (having the same remainder) is of particular importance. Steinard and Munro (1971) are quoted in Waser and Flynn (1982) by defining modular arithmetic:

> If m is a positive integer, then any two integers N and M are congruent; modulo m, if and only if there exists an integer K such that N - M = Km or
>
> $$N \bmod m \equiv M \bmod m,$$
>
> where m (a positive integer) is called the modulus.

In other words, the modulus is the quantity of numbers within which a computation takes place. That is:

$$\{0, 1, 2, 3, \ldots, m - 1\}$$

3.1.1.1 Example

If m = 256 and M = 258, N = 514, then:

514 mod 256 = 2

and

258 mod 256 = 2

This proves that M and N are congruent for that particular modulus configuration. Furthermore:

514 - 258 = K.256
K = 1

Therefore, K = 1.

***

3.1.1.2 Properties

Waser and Flynn (1982, p. 3) state that congruence has the same properties with respect to the operations of addition, subtraction, and multiplication, or any combination. In a mathematical sense (Waser and Flynn, 1982):

If N = N' mod m and M = M', then

(N + M) mod m ≡ (N' + M') mod m
(N - M) mod m ≡ (N' - M') mod m
(N * M) mod m ≡ (N' * M') mod m

3.1.1.3 Example

If m = 4, N' = 11, N = 3, M' = 5, M = 1, then:

(3 + 1) mod 4 ≡ (11 + 5) mod 4 ≡ 0
(3 - 1) mod 4 ≡ (11 - 5) mod 4 ≡ 2
(3 * 1) mod 4 ≡ (11 * 5) mod 4 ≡ 3

***

Waser and Flynn (1982) state that for modulus operations, the usual convention is to choose the least, positive residue (including zero). The following case illustrates this point:

-7 mod 3 ≡ -1 or +2

Abiding by the convention for modulus operations, the valid answer is +2.

Classically, division is defined as follows:

$$\frac{a}{b} = q + \frac{r}{b}$$

where q is the quotient and r is the remainder. However, modulus division does not extend as simply as the other three operations. For instance:

$$\frac{3}{1} \neq \frac{11}{5} \bmod 4$$

Nevertheless, division is an important operation in modular arithmetic. Waser and Flynn (1982) state that for any modulus division M/m, there is a unique quotient-remainder pair and the remainder has one of the m possible values 0, 1, 2, ..., m - 1. This leads to the notion of residue class.

A residue class, as defined by Waser and Flynn (1982), is the set of all integers having the same remainder upon division by the modulus m. For example, if m = 4, then the numbers 1, 5, 9, 13... are of the same residue class. Exactly m residue classes exist, and each integer belongs to one and only one residue class. Thus, the modulus m partitions the set of all integers into m distinct and disjoint subsets called residue classes.

3.1.1.4 Example

If m = 4, then there are four residue classes which partition the integers:

{..., -8, -4, 0, 4, 8, 12, ...}
{..., -7, -3, 1, 5, 9, 13, ...}
{..., -6, -2, 2, 6, 10, 14, ...}
{..., -5, -1, 3, 7, 11, 15, ...}

***

3.1.2 Real and rational numbers

According to Waser and Flynn (1982), real numbers also need to be represented in a machine with the limitation of finitude. This is achieved by approximating real and rational numbers, by terminating sequences of digits. Therefore, all numbers (real, rational, and integers) can be operated on as if they were integers. This can be done under the assumption that scaling and rounding are done properly.

3.1.3 Integers

Integers can be represented by positional weight. Waser and Flynn (1982) state that in a weighted positional system, the number N is the sequence of m + 1 digits d_m, d_{m-1}, ..., d_2, d_1, d_0, which in base, or radix, b can be calculated to give $N = d_m b^m + d_{m-1} b^{m-1} + \ldots d_1 b + d_0$. The digit values for d_i may be any integer between 0 and b - 1.

3.1.3.1 Example

In the familiar decimal system, the base is b = 10, and the 4-digit number 1736 is:

$N = 1736_{10} = (1 * 10^3) + (7 * 10^2) + (3 * 10^1) + 6$

Similarly, for the binary system b = 2, a 5-digit number 10010 is equivalent to:

$N = (1 * 2^4) + (0 * 2^3) + (1 * 2^1) + 0 = 18_{10}$

The leading digit, d_m, is the most significant digit, or the most significant bit for the decimal and binary systems, respectively. Likewise, d_0 is designated as the least significant digit or bit.

***

Lacking from the above definition of an integer are negative numbers. Garner (1965), cited in Waser and Flynn (1982, p. 6), describes the more commonly known concepts to represent signed numbers:

1. Magnitude plus sign: Digits are represented according to the simple positional number system. An additional high-order symbol represents the sign. This code is natural for humans but unnatural for a modular computer system.
2. Complement codes: Two types are commonly used, namely, radix compliment code and diminished radix complement code. Compliment coding is natural for computers, since no special sign symbology or computation is required. In binary arithmetic, the RC code is called two's complement, and the DRC is called the one's compliment.

Complement codes will be described further because of their wider use in arithmetic systems.

3.1.3.2 Radix complement code

Waser and Flynn (1982) explain radix complement codes. Suppose N is a positive integer of the form:

$$N = d_m b^m + d_{m-1} b^{m-1} + \ldots d_1 b + d_0$$

The maximum value N may assume is $b^{m+1} - 1$. Thus, $b^{m+1} > N \geq 0$.

To represent -N, the radix complement of N must be defined:

$$RC(N) = b^{m+1} - N$$

For ease of representation, let $n = m + 1$. Substituting n into RC(N) gives:

$$RC(N) = b^n - N$$

Assume b is even and suppose M and N are n-digit numbers. The calculation M - N can be accomplished using the addition operation. M and N may be either positive or negative numbers so long as:

$$\frac{b^n}{2} - 1 \geq M, N \geq -\frac{b^n}{2}$$

Then

$$M - N \equiv (M - N) \bmod b^n$$

and

$$(M - N) \bmod b^n = (M \bmod b^n - N \bmod b^n) \bmod b^n$$

If -N is replaced by $b^n - N$, the equality remains unchanged. That is, by taking:

$$(M - N) \bmod b^n = (M \bmod b^n - (b^n - N) \bmod b^n) \bmod b^n$$
$$= M \bmod b^n - N \bmod b^n$$

The complement of b^n - N can be derived easily:

For $N < b^n$, let N be represented as $X_m \ldots X_0$. The operation b^n - N can thus be represented as follows:

$$\begin{array}{r} 10\,000\ldots0 \\ -\ \underline{X_m XXX_i \ldots X_0} \end{array}$$

where m = n - 1

The radix compliment of any digit X_i is designated $RC(X_i)$. For all lower order digits which satisfy

$X_0 = X_1 = \ldots = X_i = 0$

the $RC(X_i)$ is

$RC(X_0) = RC(X_1) = \ldots = RC(X_i) = 0$

For $X_{i+1} \neq 0$, the first (lower order) nonzero element in N

$RC(X_{i+1}) = b - X_{i+1}$

For all elements X_j thereafter, $m \geq j \geq i + 2$:

$RC(X_j) = b - 1 - X_j$

As an example, in a three-position decimal number system, the radix complement of the positive number 245 is 1000 - 245 = 755. This illustrates that by properly scaling the represented positive and negative numbers about zero, no special treatment of the sign is required. Therefore, in radix complement code, the most significant digit indicates the sign of the number. In the base 10 system, the digits 5, 6, 7, 8 and 9 (in the most significant position) indicate negative numbers. That is, the three digits represent numbers from +499 to -500. In the binary system, the digit 1 is an indication of negative numbers.

3.1.3.2.1 Example

If M = +250, N = +245, then M - N is:

$$\begin{array}{rr} & 250 \\ - & \underline{245} \end{array} \quad \Rightarrow \quad \begin{array}{rr} & 250 \\ + & \underline{755} \\ & 1005 \end{array} \quad \text{mod } 1000 \equiv 5$$

***

Matula (cited in Kuczborski, 1993, p. 40) asserts the critical aspects of radix systems, which are:

1. Completeness of radix representation. That is, the ability to represent all possible values within a specific range.
2. Uniqueness of radix representation. Each value should be represented by a unique string of digits.
3. Sign detection.
4. Representation of zero.
5. Carry propagation from less to more significant positions for addition.

Example 3.1.3.2.2 illustrates how conventional linear weighted number operations are performed in RC.

3.1.3.2.2 Example

Perform the following operation in decimal and binary: 19 486 - 22 139.

Decimal calculation

The calculation can be performed using RC.

```
= 19 486 + (100 000 - 22 139)
= 19 486 + 77 861

          19 486
  +       77 861
    ------------
          97 347

97 347 = 97 347 - 100 000
= -2 653
```

Binary calculation

The calculation can be performed using RC.

```
= 0100 1100 0001 1110 + (1 0000 0000 0000 0000 - 0101 0110 0111 1011)
= 0010 0101 0000 1110 + 1010 1001 1000 0101

            0100 1100 0001 1110
  +         1010 1001 1000 0101
    ------------------------------
            1111 0101 1010 0011

1111 0101 1010 0011 = 1111 0101 1010 0011 - 1 0000 0000 0000 0000
= -0000 1010 0101 1101
= -2 653
```

***

As is evident from Example 3.1.3.2.2, carry propagation hinders the speed of the calculation. Addition is forced to be performed in a serial manner, uncovering further complications. That is, the time the addition takes to complete is dependent upon the wordlength of both operands. Waser and Flynn (1982, p. 54) point out that in the conventional linear weighted number system, an operation on long words is slower due to the carry propagation.

Kuczborski (1993) explains that if carry propagation is restricted to a single digit position, then the following objectives are achievable:

1. Parallel processing of all digit positions.
2. Fast result rounding.
3. A higher degree of circuit reliability.
4. More regular very large scale integration designs with local communications.

For some real-time systems, a real-time arithmetic system is required. For example, in real-time morphological image processors, the computation times need to be kept constant. Morphological image processors employ mathematical morphology to achieve image manipulation. The two basic operations of mathematical morphology are addition, and magnitude comparison. This project focused on creating VLSI circuits using a nonconventional number scheme that could add and compare numbers very efficiently. Multiplication using the same number scheme was also investigated. However, the hybrid nonconventional number system that is discussed later on may also be highly applicable in areas where the ability to process very large numbers is required. Example applications include data encryption, speech analysis and recognition, and image compression.

Nonconventional digit representations were analysed to see if the above objectives are attainable. The nonconventional representations discussed in the following sections are SDNR and RNS. The benefits from combining these two number representations is discussed after both are treated separately.

3.1.4 Signed Digit Number Representation

SDNRs are weighted number systems. They are also redundant. It is this redundancy which limits carry propagation to one position to the left during the operations of addition and subtraction. This, in turn, allows for parallel arithmetic only when a certain condition, known as the threshold value, is met. An important characteristic of SDNRs is that better efficiency is achieved, in terms of processing and storage requirements, when larger radices are used.

Avizienis (1961) describes SDNRs by comparing them to the conventional number system:

> In a conventional number representation with an integer radix r > 1, each digit is allowed to assume exactly r values, that is, 0, 1, ..., r - 1. In a redundant representation with the same radix r, each digit is allowed to assume more than r values.

Kuczborski (1993) states that in SDNR, an integer is represented by the digit string:

$a_n a_{n-1} \ldots a_1 a_0$

The value of this digit string is determined by:

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

where A = SDNR number.
a = negative, zero, or positive digit.
r = radix (positive integer).

Furthermore, the magnitude of the digit must be set within the range:

$r + 2 \leq n \leq 2r - 1$

where n = digit magnitude.

This range restriction creates several desirable algebraic properties (Kuczborski, 1993):

1. The lower bound of n limits carry propagation to a single position, resulting in fully parallel addition and subtraction.
2. The lower bound includes the weaker condition of completeness ($n \geq r$).
3. The upper bound of n ensures that the sign of an SDNR number equals the sign of its most significant non-zero digit.
4. The upper bound guarantees a unique representation of zero.

An implication of $r + 2 \leq n \leq 2r - 1$, as pointed out by Kuczborski (1993) is that:

$r > 2$

SDNR allows two types of digit sets. The first types are asymmetric about zero and are of the form:

$\{-a, -a + 1, \ldots, -1, 0, 1, \ldots, b - 1, b\}$

where a = positive digit.
b = positive digit.
$a \neq b$

The other type of digit sets are symmetric about zero and can be represented as follows:

$\{-a, -a + 1, \ldots, -1, 0, 1, \ldots, a - 1, a\}$

where a = positive digit.

The latter type of digit set is preferable because they allow easier handling of negative numbers (Kuczborski, 1993). This project concentrated on symmetric digit sets.

Choosing the digit set can now be addressed. Note that the choice of digit set has an effect on the degree of redundancy implied in the SDNR. Kuczborski (1993) says that for a minimal redundant digit set:

$$a = \text{FLOOR}\left(\frac{r}{2}, 1\right) + 1$$

On the other hand, for a maximum redundant digit set (Kuczborski, 1993):

$a = r - 1$

3.1.4.1 Example

For radix 10:

Minimum redundancy:

$$a = \text{FLOOR}\left(\frac{r}{2}, 1\right) + 1$$

$$= \text{FLOOR}\left(\frac{10}{2}, 1\right) + 1$$

$= 6$

Maximum redundancy:

$a = r - 1$
$= 10 - 1$
$= 9$

Therefore, the radix 10 minimally redundant digit set = {-6, -5, ..., -1, 0, 1, ..., 5, 6}
In comparison, the radix 10 maximally redundant digit set = {-9, -8, ..., -1, 0, 1, ..., 8, 9}

***

Kuczborski (1993) reports that small radices utilise data storage inefficiently. For example, a 16-bit radix-4 SNDR system has a relatively small range when compared with the equivalent conventional 16-bit two's compliment representation. Example 3.1.4.2 illustrates this point.

3.1.4.2 Example

For 16-bit radix-4 SDNR system:

r = 4
n = 16

For minimum redundancy:

$$a = FLOOR\left(\frac{r}{2},1\right) + 1$$

$$= FLOOR\left(\frac{4}{2},1\right) + 1$$

$= 3$

(a is the same for maximum redundancy)

$D = 2a + 1$
$= 2(3) + 1$
$= 7$

$N = CEILING(\log_2 D, 1)$
$= CEILING(\log_2(7), 1)$
$= 3$

$$A = INT\left(\frac{n}{N}\right)$$

$$= INT\left(\frac{16}{3}\right)$$

$= 5$

$$R = \{-(a*r^{(A-1)} + a*r^{(A-2)} + \ldots + a*r^1 + a*r^0) \ldots +(a*r^{(A-1)} + a*r^{(A-2)} + \ldots + a*r^1 + a*r^0)\}$$

$$\therefore R = \{-(3*4^4 + 3*4^3 + \ldots + 3*4^1 + 3*4^0) \ldots +(3*4^4 + 3*4^3 + \ldots + 3*4^1 + 3*4^0))\}$$
$= \{-1023 \ldots +1023\}$

For conventional 16-bit two's compliment representation:

$$R = \{-2^{(n-1)} \ldots +2^{(n-1)} - 1\}$$

where n = word length (bits).
= 16

$$\therefore R = \{-2^{(16-1)} \ldots +2^{(16-1)} - 1\}$$
$= \{-32\ 768 \ldots +32\ 767\}$

where D = number of digits in digit set.
N = number of bits required per digit.
A = number of allowable digits.
n = word length.
R = range.

Therefore, the range for the 16-bit radix-4 SDNR system = {-1023 ... +1023}
In comparison, the range for the conventional 16-bit two's compliment representation = {-32 768 ... +32 767}

***

Theoretically, higher radices widen the dynamic range of the data, speed up operations such as multiplications, and reduce the silicon area for interconnections (routing complexity).

An SDNR arithmetic system must be able to communicate with the external environment. Therefore, assuming the conventional binary number system is used externally, generalised conversion procedures are required to translate conventional binary numbers to SDNR, and vice versa.

3.1.4.3 Radix conversion

Before translation between the conventional number system and SDNR can take place, both systems must have the same radix base. In a sense, one number system must be manipulated to be made "compatible" with the other. This is an important initial step. Usually the conventional number system will have a different radix to that of the target SDNR system. As mentioned previously, if the magnitude of the radix used in the SDNR is increased, the dynamic range of the data increases, operations like multiplication are accelerated, and the number of processing elements is decreased. Therefore, in an ideal situation, the arithmetic system should be able to accept conventional binary numbers (radix 2) from an external source, and use a large internal radix representation for SDNR.

For the arithmetic system, the external number system (the conventional system) requires it's base to be changed in order to match the internal SDNR radix. When the externally sourced number is loaded into the arithmetic system, it must then be converted to the internal radix representation. Correspondingly, when the arithmetic system completes the specified operation on the number(s), the result must be reconverted back into the radix of the conventional number system.

Once the required radix conversion has taken place, the conventional number system (external arithmetic system input) has to be converted to SDNR notation (for internal arithmetic system processing). This topic is discussed in the next section.

3.1.4.4 Conventional number systems to SDNR conversion

Translation from conventional number systems into SDNR requires several algorithmic steps. The algorithm presented in this section is an adaptation of a radix-r SDNR adder described in Kuczborski (1993). To begin with, a threshold sum value, which will determine carry values, must be defined within the range:

$$1 \leq r - a \leq t \leq a - 1$$

where t = threshold sum value.
r = radix.
$\{-a \ldots 0 \ldots +a\}$

The conversion procedure treats each conventional digit i separately during conversion. The algorithm is as follows:

CONVERT_CONVENTIONAL_TO_SDNR_STAGE_1

INPUTS: X_i
OUTPUTS: INTERMEDIATE_CONVERSION$_i$, C_{i+1}

BEGIN

IF $X_i > t$ **THEN**
$C_{i+1} = 1$

```
        IF X_i < t THEN
                C_i+1 = -1
        ELSE
                C_i+1 = 0
        ENDIF
        INTERMEDIATE_CONVERSION_i = X_i - rC_i+1

END

CONVERT_CONVENTIONAL_TO_SDNR_STAGE_2

INPUTS: INTERMEDIATE_CONVERSION_i, C_i
OUTPUTS: CORRECTED_CONVERSION_i

BEGIN

        CORRECTED_CONVERSION_i = INTERMEDIATE_CONVERSION_i + C_i

END
```

where t = threshold sum value.
r = radix.
X_i = conventional radix r-digit in $X_nX_{n-1} \ldots X_1X_0$.
C_{i+1} = carry out.
C_i = carry in.
$INTERMEDIATE\_CONVERSION_i$ = SDNR/RNS radix-r intermediate conversion for X_i.
$CORRECTED\_CONVERSION_i$ = SDNR/RNS radix-r corrected conversion for X_i.

Example 3.1.4.4.1 shows how a radix-r conventional number system can be converted into radix-r SNDR notation.

3.1.4.4.1 Example

Convert 1996_{10} to SDNR notation with the following SDNR characteristics:

r = 10

$$a = FLOOR\left(\frac{r}{2},1\right) + 1$$

$$= FLOOR\left(\frac{10}{2},1\right) + 1$$

= 6

t = a - 1
= 6 - 1
= 5

Valid digit set = {-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6}

1	9	9	6	
1	$\bar{1}$	$\bar{1}$	$\bar{4}$	STAGE 1
1	1	1		STAGE 2
2	0	0	$\bar{4}$	

Therefore, $1996_{10} = 200\bar{4}_{SDNR10}$

***

Once the required processing has taken place, for example, adding two operands, the SDNR data has to be converted back to conventional binary form.

3.1.4.5 SDNR to conventional number systems conversion

Avizienis (1961) suggests several methods for converting SDNR to a conventional representation. The first approach is to consider the SDNR number as the sum of two numbers in conventional representation of the same length, one of which is positive and the other negative. Negative and positive digits are separated to form these two numbers, which then can be summed in a conventional representation adder circuit, resulting in the desired conventional representation. The second approach implies a serial conversion. That is, the conversion process traverses the SDNR number, from LSD to the MSD, until all SDNR digits have been translated into their equivalent conventional form.

Ercegovac and Lang (1987) provide an alternative to the methods described by Avizienis (1961). They take an on-line approach, stating that SDNR to conventional conversion can be performed efficiently without carry-propagate addition using an on-the-fly method. The algorithm Ercegovac and Lang (1987) propose has the following characteristics:

1. It performs conversion on the fly, as the digits of the result are obtained in a serial fashion from most to least significant.
2. It uses conditional logic. That is, it uses two conditional forms of the current result.
3. It has a delay which is roughly equal to two logic levels plus a register shift/load time.

The algorithm devised by Ercegovac and Lang (1987) for on-the-fly conversion is part of a greater area of study known as on-line arithmetic.

3.1.4.6 SDNR arithmetic

The operations of addition, subtraction, shifting, multiplication, division, and sign and overflow detection can be performed in SDNR notation. The following section explain the various SDNR arithmetic operations. The focus is on addition, sign detection, and overflow detection.

3.1.4.7 SDNR operations

The elementary operation of addition in SDNR is shown in Example 3.1.4.7.1.

3.1.4.7.1 Example

Add $30\bar{2}5_{SDNR10}$ and $2\bar{3}32_{SDNR10}$ based on the following SDNR attributes:

$r = 10$

$$a = \text{FLOOR}\left(\frac{r}{2},1\right) + 1$$

$$= \text{FLOOR}\left(\frac{10}{2},1\right) + 1$$

$= 6$

$t = a - 1$
$= 6 - 1$
$= 5$

3	0	$\bar{2}$	5	
2	$\bar{3}$	3	2	
5	$\bar{3}$	1	$\bar{3}$	STAGE 1
		1		STAGE 2
5	$\bar{3}$	2	$\bar{3}$	

Therefore, $30\bar{2}5_{SDNR10} + 2\bar{3}32_{SDNR10} = 5\bar{3}2\bar{3}_{SDNR10}$

***

An explanation of SDNR multiplication and division can be found in Avizienis (1961).

Sign detection in SDNR is relatively simple. This is due to the fact that each negative and positive digit in SDNR is identified by a unique symbol. Therefore, as pointed out by Kuczborski (1993), the sign of a SDNR number can be determined from the sign of the most significant non-zero digit in that particular number.

Overflow is discussed in Spaniol (1981). He presents an overflow detection and correction system for SDNRs. The concept of overflow in SDNR can be realised by considering the following generalised case:

	A_n	A_{n-1}	A_{n-2}	...	A_1	A_0
	u_n	u_{n-1}	u_{n-2}	...	u_1	u_0
	T_n	T_{n-1}	T_{n-2}	...	T_1	T_0
(C_n)	c_{n-1}	c_{n-2}	c_{n-3}	...	c_0	c_{-1}
(S_{n+1})	S_n	S_{n-1}	S_{n-2}	...	S_1	S_0

Spaniol (1981) states that if $S_n \neq 0$, then an overflow condition exists. Position S_{n+1} does not need to be calculated. The overflow may be corrected if the two MSDs in $S_nS_{n-1}...S_0$, which differ from zero, have different signs. If the sum has the form:

$S_nS_{n-1}...S_0 = S_n0...0S_{n-k-1}S_{n-k-2}...S_0$

where k = number of digits required to represent $0...0$ $(k \geq 0)$.
$S_n, S_{n-k-1} \neq 0$
$SIGN(S_n) \neq SIGN(S_{n-k-1})$

then this overflow situation can be corrected to:

$0S_{n-1}'...S_0'$

where $S_i' = r - 1;$ $\quad i = n-1, ..., n-k$
$r + S_i;$ $\quad i = n-k-1$ when $S_n = 1$, $S_{n-k-1} < 0$
$S_i;$ $\quad i = n-k-2, ..., 0$

or $S_i' = \bar{d} - 1;$ $\quad i = n-1, ..., n-k$
$\bar{d} + S_i;$ $\quad i = n-k-1$ when $S_n = 1$, $S_{n-k-1} > 0$
$S_i;$ $\quad i = n-k-2, ..., 0$

Therefore, as pointed out by Spaniol (1981), an overflow situation is indicated by $S_n \neq 0$, and is correctable if the number can be represented with n digits. That is, if the next position different from zero in $S_{n-1}...S_0$ has a sign different from S_n.

However, automatic correction of all overflow situations is performed at the expense of either cycle time or hardware complexity (Spaniol, 1981). If the value of k is bounded, then overflow correction can be simplified. In other words, if the length k of the zero block following S_n does not exceed the fixed value of k*, then cycle time or hardware complexity can be reduced. The simplest case where k* = 0 will be considered. Spaniol (1981) extends the generalised case, stated previously, so that it allows for overflow correction for k* = 0:

A_n	A_{n-1}	A_{n-2}	...	A_1	A_0
u_n	u_{n-1}	u_{n-2}	...	u_1	u_0
T_n	T_{n-1}	T_{n-2}	...	T_1	T_0
c_{n-1}	c_{n-2}	c_{n-3}	...	c_0	c_{-1}
S_n	S_{n-1}	S_{n-2}	...	S_1	S_0
ε_n	ε_{n-1}				
S_n'	S_{n-1}'	S_{n-2}'	...	S_1'	S_0'

where $\varepsilon_n = 0$ when $S_n = 0$ or $S_{n-1} = 0$

$\overline{S_n}$ otherwise

$\varepsilon_{n-1} = -r\varepsilon_n$

$= 0$ when $S_n = 0$ or $S_{n-1} = 0$

rS_n otherwise

An overflow remains uncorrected if $S_n' \neq 0$. Spaniol (1981) states that if the same probability applies to all digits A_i and u_i, and if they are not interdependent, then:

$$P(S_n' \neq 0) = \frac{1}{4r - 1}$$

Cases of overflow detection and correction are illustrated in Example 3.1.4.7.2.

3.1.4.7.2 Example

Add $5\bar{2}3\bar{3}_{SDNR10}$ and $14\bar{2}1_{SDNR10}$ based on the following SDNR attributes:

r = 10

$$a = \text{FLOOR}\left(\frac{r}{2}, 1\right) + 1$$

$$= \text{FLOOR}\left(\frac{10}{2}, 1\right) + 1$$

= 6

t = a - 1
= 6 - 1
= 5

	5	$\bar{2}$	3	$\bar{3}$	
	1	4	$\bar{2}$	1	
	$\bar{4}$	2	1	$\bar{2}$	STAGE 1
1	0				STAGE 2
1	$\bar{4}$	2	1	$\bar{2}$	
$\bar{1}$	r				OVERFLOW CORRECTION
	6	2	1	$\bar{2}$	

Therefore, $5\bar{2}33_{SDNR10} + 14\bar{2}1_{SDNR10} = 621\bar{2}_{SDNR10}$

***

The RNS is the second of the nonconventional number systems to be investigated. It is described in the following section.

3.1.5 Residue Number System

Unlike SDNR, the RNS is a nonweighted number system. Due to the absence of carry propagation in RNS, it is possible to perform pure parallel arithmetic under any condition.

The RNS is an extension of modular arithmetic discussed in section 3.1.1-Modular arithmetic. Kuczborski (1993) states that the residue representation of an integer I is an n-tuple

$$<I_1, I_2, \ldots, I_n>$$

related to another n-tuple of relatively prime integer moduli p_i:

$$(p_1, p_2, \ldots, p_n)$$

Kuczborski (1993) asserts that RNS maintains a complete and unique representation within a finite dynamic range defined by:

$$M = p_1 p_2 \ldots (p_n - 1)$$

3.1.5.1 Selection of moduli

The most important consideration when designing RNS systems is the choice of the moduli set $(p_1, p_2, \ldots, p_n)$. According to Abdallah and Skavantzos (1995), the moduli p_is should satisfy the following:

1. They should be relatively prime. That is, there should be no common divisor between any of the moduli in the set $(p_1, p_2, \ldots, p_n)$.
2. The moduli p_is should be as small as possible so that operations modulo p_i require minimum computational time.
3. The moduli p_is should imply simple weighted to RNS and RNS to weighted conversions as well as simple RNS arithmetic.
4. The product of the moduli should be large enough in order to implement the desired dynamic range.
5. The moduli p_is should create a balanced decomposition of the dynamic range. That is, the differences between the number of bits to represent the different moduli should not be very large.

Abdallah and Skavantzos (1995) have classified moduli-set choices that have been considered by RNS researchers. They are:

1. Sets of the form $(2^n - 1, 2^n, 2^n + 1)$, where n is a positive integer. These moduli imply simple conversions, simple RNS arithmetic, and balanced decomposition of the dynamic range. However, if large dynamic ranges are required, then the sizes of such moduli become large and the performance of the system degrades.
2. Sets where all the moduli are Mersenne or Fermat numbers, while only one modulus is of the form 2^n. Such choices can result in unbalanced dynamic range decomposition.
3. Sets with many arbitrary small-size prime moduli and only one modulus of the form 2^n. ROM table lookups should be used, as the choice of arbitrary moduli may imply complex conversions and RNS arithmetic. However, the cost of such ROM-based RNS systems could be prohibitive.

Just like with SDNR arithmetic systems, RNS systems require conventional to RNS procedures if the system is to perform operations on the input data. If an external system requires the processed information, then RNS to conventional conversion circuits are also needed.

3.1.5.2 Conventional number systems to RNS conversion

An integer value I is converted into its RNS equivalent by modulo operations (Kuczborski, 1993):

$$<I \bmod p_1, I \bmod p_2, \ldots, I \bmod p_n>$$

Example 3.1.5.2.1 illustrates this procedure.

3.1.5.2.1 Example

Convert 29_{10} to a residue number with the moduli set (5, 3, 2).

$M = p_1 p_2 \ldots (p_n - 1)$
$= (5)(3)(2)$
$= 30$

30 unique values can be represented by the moduli set (5, 3, 2).

$29_{10} = <29 \bmod 5, 29 \bmod 3, 29 \bmod 2>$
$= <4, 2, 1>$

Therefore, $29_{10} = <4, 2, 1>_{RNS}$

***

3.1.5.3 RNS to conventional number systems conversion

Taylor (1984) states that conversion of a RNS number into its radix equivalent can be achieved through the Chinese Remainder Theorem:

$$I = \left(\sum_{i=1}^{n} s_i \left(X_i \, s_i^{-1} \bmod p_i \right) \right) \bmod M$$

where $M = p_1 p_2 \ldots p_n$

$$s_i = \frac{M}{p_i}$$

$$\left(s_i^{-1} s_i\right) \bmod p_i = 1$$

A more straightforward and faster method of conversion is to use lookup tables. Lookup tables can be used to perform the RNS to conventional number system conversion, and vice versa.

3.1.5.4 RNS arithmetic

As a result of it's carry free nature, the RNS is suitable for addition, subtraction, and multiplication. The weaknesses of RNS, as pointed out by Kuczborski (1993), include operations such as sign detection, magnitude comparisons, overflow detection, and division. These weaknesses are primarily due to the fact that the RNS is an nonweighted number system.

In the next section, the efficient RNS operations of addition, subtraction, and multiplication will be discussed.

3.1.5.5 RNS operations

According to Kuczborski (1993), RNS benefits greatly from it's inherent parallelism. This means that, because of the carry-free nature of RNS, additions, subtractions, and multiplications can be calculated on an independent, digit by digit basis (Kuczborski, 1993):

$$\langle I_1, I_2, \ldots, I_n \rangle \blacksquare \langle J_1, J_2, \ldots, J_n \rangle = \langle (I_1 \blacksquare J_1) \bmod p_1, (I_2 \blacksquare J_2) \bmod p_2, \ldots, (I_n \blacksquare J_n) \bmod p_n \rangle$$

where $\blacksquare$ = addition, subtraction, or multiplication.

Example 3.1.5.5.1 illustrates RNS addition, subtraction, and multiplication.

3.1.5.5.1 Example

Moduli set is (5, 3, 2).

9		<4, 0, 1>
+ 16	→	<1, 1, 0>
25		<0, 1, 1>
19		<4, 1, 1>
- 8	→	<3, 2, 0>
11		<1, 2, 1>
7		<2, 1, 1>
* 4	→	<4, 1, 0>
28		<3, 1, 0>

***

Before moving onto the hybrid SDNR/RNS scheme, a comparison of the two nonconventional number systems is presented in the next section.

3.1.6 Signed Digit Number Representation/Residue Number System comparison

Taylor (1984) states that in general, an integer X, which has a fixed-radix, weighted-number representation, with respect to a radix r, is given by:

$$X = \sum_{i=0}^{n-1} a_i r^i$$

where $a_i \in Z_r$

The number of integer values of X that possess an n-digit, fixed-radix representation are r over the range $[0, r^{n-1}]$. Notice that this definition of a fixed-radix, weighted-number system representation is similar to the SDNR definition. Thus, SDNR notation can be regarded as a fixed-radix, weighted-number representation. Taylor (1984) describes some of the favourable characteristics of a fixed-radix system as being:

1. Algebraic comparison.
2. Dynamic range extension. That is, more digits can be added to increase the range.
3. Multiplication and division by simple arithmetic shifts. This is not so simple when using SDNR.
4. Simplified overflow and sign-detection.

Taylor (1984) points out that the disadvantage of the fixed-radix, weighted-number system is that carry information must be propagated from the LSD to the MSD. SDNR minimises this problem by restricting carry propagation to one position by introducing redundancy into the number system.

While SDNR minimises carry propagation, the RNS eliminates it altogether. That is, the RNS is a carry-free system and is potentially very fast for certain problems, even though the advantages of the fixed-radix system do not carry over. The advantages of the RNS are addition, subtraction, and multiplication operations. The disadvantages of the RNS are inherently complex algebraic comparisons, overflow and sign detection, and division.

3.1.7 Signed Digit Number Representation/Residue Number System

Kuczborski (1993) states that the idea of combining the SDNR with the RNS is based on the natural parallelism of the latter representation. By assigning word level operations to the SDNR and digit level operations to the RNS, the disadvantages of both number system can be overcome.

By combining the SDNR and RNS number systems, two views of the representation become apparent. At the word level, the SDNR/RNS word is represented in the SDNR domain. At the digit level, the SDNR/RNS word is represented in the RNS domain. In effect, the RNS decomposes the chosen SDNR digit set, so that a digital designer can create logic blocks which are smaller, faster, and more manageable.

Kuczborski (1993) points out that the use of the RNS for coding the SDNR digits requires two disjunctive sets for positive and negative values. An odd product of all n moduli has a symmetric range of (Kuczborski, 1993):

$$\frac{-(p_1 p_2 \cdots p_n - 1)}{2} \cdots \frac{(p_1 p_2 \cdots p_n - 1)}{2}$$

For an even product, the range becomes:

$$-\left(p_1 p_2 \cdots \frac{p_n}{2}\right) \cdots \left(p_1 p_2 \cdots \frac{p_n}{2} - 1\right)$$

The RNS representation of any SDNR digit is, according to Kuczborski (1993):

$i = \langle X_1, X_2, \ldots, X_n \rangle$

where $i \in \{-a, -a + 1, \ldots, -1, 0, 1, \ldots, a - 1, a\}$

The RNS representation can be determined as follows:

$X_1 = i \bmod p_1$
$X_2 = i \bmod p_2$
...
$X_n = i \bmod p_n$

SDNR/RNS arithmetic is quite straightforward. SDNR arithmetic algorithms discussed in section 3.1.4.7-SDNR operations are used at the word level to perform addition, subtraction, overflow and sign detection. At the digit level, addition, subtraction, and multiplication can take place by using the following RNS arithmetic rule:

$Zi = (X_i \blacksquare Y_i) \bmod p_i$

where $i = 1, 2, \ldots, n$
$\blacksquare$ = addition, subtraction, or multiplication.

The operations of SDNR/RNS addition (and subtraction), and multiplication are discussed in the following sections.

3.1.7.1 SDNR/RNS addition

An SDNR/RNS integer is represented differently at different levels. At the word level, the number can be treated as a SDNR integer. At the digit level, each digit can be viewed upon as a RNS number. The SDNR/RNS addition algorithm is used at the word level. Thus, the SDNR algorithm for addition will be used to add two SDNR/RNS integers. As it will become clear later, for high radices and numerical ranges, RNS addition at the digit level speeds up computations.

Before SDNR/RNS addition can take place, however, an SDNR/RNS configuration must be chosen. The main constraint is that the configuration must be able to represent the conventional integer operands. For the case of the adder system in section 4.2-Case studies, the requirement is that it must be able to add two 64-bit conventional integers. Therefore, the SDNR/RNS configuration chosen must be able to represent a range from 0 to $(2^{64} - 1) = 0$ to $1.84 * 10^{19}$, for the case of unsigned integers, relatively efficiently. An analysis in choosing an optimal SDNR/RNS configuration is given in section 4-Analysis. Guidelines for choosing a radix, digit set, and RNS moduli set are given in that section.

The steps required to perform SDNR/RNS addition are as follows:

1. During conventional to SDNR/RNS conversion, if the sign of the conventional integer is negative, then toggle sign of each SDNR/RNS digit.
2. Choose a threshold value t to satisfy:
   $1 \le r - a \le t \le a - 1$
   where t = threshold sum value.
   r = radix.
   Usually, $t = a - 1$.
3. Execute adder algorithms:

SDNR_RNS_ADD_STAGE_1

INPUTS: X_i, Y_i
OUTPUTS: INTERMEDIATE_SUM$_i$, C_{i+1}

BEGIN

IF $X_i + Y_i > t$ **THEN**
$C_{i+1} = 1$
IF $X_i + Y_i < t$ **THEN**
$C_{i+1} = -1$
ELSE
$C_{i+1} = 0$
ENDIF
INTERMEDIATE_SUM$_i = X_i + Y_i - rC_{i+1}$

END

SDNR_RNS_ADD_STAGE_2

INPUTS: INTERMEDIATE_SUM$_i$, C_i
OUTPUTS: CORRECTED_SUM$_i$

BEGIN

CORRECTED_SUM$_i$ = SUM$_i$ + C_i

END

where t = threshold sum value.
r = radix.
X_i = conventional radix r-digit in $X_nX_{n-1} \ldots X_1X_0$.
Y_i = conventional radix r-digit in $Y_nY_{n-1} \ldots Y_1Y_0$.
C_{i+1} = carry out.
C_i = carry in.
INTERMEDIATE_SUM$_i$ = SDNR/RNS radix-r intermediate sum for $X_i + Y_i$.
CORRECTED_SUM$_i$ = SDNR/RNS radix-r corrected sum for $X_i + Y_i$.

The SDNR/RNS adder must be able to accept all possible values of INTERMEDIATE_SUM$_i$ ($X_i + Y_i$). That is, the dynamic range of the chosen moduli set must be able to represent the extended digit set:

$\{-2a, -2a + 1, \ldots, -1, 0, 1, \ldots, 2a - 1, 2a\}$

An example of SDNR/RNS addition is given in the following section.

3.1.7.1.1 Example

Add $\bar{4}\,6\,2\,\bar{1}_{SDNR10}$ and $1\,\bar{5}\,\bar{4}\,6_{SNDR10}$ based on the following SDNR/RNS attributes:

$r = 10$

$$a = \text{FLOOR}\left(\frac{r}{2},1\right) + 1 = \text{FLOOR}\left(\frac{10}{2},1\right) + 1 = 6$$

$t = a - 1 = 6 - 1 = 5$

$p1 = 3$

p2 = 5
p1p2 = 15

The addition will be performed in the SDNR domain, and then in the SDNR/RNS context.

SDNR arithmetic

4	$\bar{6}$	2	$\bar{1}$	
1	$\bar{5}$	4	$\bar{6}$	
5	$\bar{11}$	6	$\bar{7}$	INTERMEDIATE SUM
5	$\bar{1}$	$\bar{4}$	3	CORRECTED SUM
$\bar{1}$	1	$\bar{1}$	0	CARRIES
4	0	$\bar{5}$	3	

SDNR/RNS arithmetic

decimal	decimal mod p1	decimal mod p2	
0	0	0	carry = 0
1	1	1	
2	2	2	
3	0	3	
4	1	4	
5	2	0	
6	0	1	carry = 1
7	1	2	
8	2	3	
9	0	4	
10	1	0	
11	2	1	
12	0	2	

decimal	decimal mod p1	decimal mod p2	
-12	0	3	carry = -1
-11	1	4	
-10	2	0	
-9	0	1	
-8	1	2	
-7	2	3	
-6	0	4	
-5	1	0	carry = 0
-4	2	1	
-3	0	2	
-2	1	3	
-1	2	4	

<1, 4>	<0, 4>	<2, 2>	<2, 4>	
<1, 1>	<1, 0>	<1, 4>	<0, 4>	
<2, 0>	<1, 4>	<0, 1>	<2, 3>	INTERMEDIATE SUM
<2, 0>	<2, 4>	<2, 1>	<0, 3>	CORRECTED SUM
<2, 4>	<1, 1>	<2, 4>	<0, 0>	CARRIES
<1, 4>	<0, 0>	<1, 0>	<0, 3>	

$\therefore \bar{4}\,6\,2\,\bar{1}_{SDNR10} + 1\,5\,4\,\bar{6}_{SNDR10} = 40\,\bar{5}\,3_{SDNR10}$

***

There are two techniques available to satisfy the dynamic range required by the extended digit set. The first is to use disjoint digit sets, and the second makes use of nondisjoint digit sets.

3.1.7.1.2 Disjoint digit sets

A satisfactory RNS moduli set dynamic range results in a disjunctive sets for positive and negative intermediate sums. The condition for disjoint sets is as follows:

$4a + 1 \leq p_1p_2...p_n$

Figure 2 depicts both disjoint sets and also:

1. Shows RNS codes for the INTERMEDIATE_SUM$_i$ = $X_i + Y_i$.
2. Specifies carry values for various ranges of the INTERMEDIATE_SUM$_i$.
3. Identifies possible sign combinations of X_i and Y_i.

	INTERMEDIATE _SUM$_i$ (SDNR)	INTERMEDIATE_SUM$_i$ (RNS)
carry = 0 ++, +-	0 1 ⋮ a - 1	< 0 mod p1, 0 mod p2, ..., 0 mod pn > < 1 mod p1, 1 mod p2, ..., 1 mod pn > ⋮ < (a - 1) mod p1, (a-1) mod p2, ..., (a-1) mod pn >
carry = 1 ++	a ⋮ 2a	< a mod p1, a mod p2, ..., a mod pn > ⋮ < 2a mod p1, 2a mod p2, ..., 2a mod pn >
carry = -1 -- --, +-	-2a ⋮ -a	< (p1p2...pn - 2a) mod p1, (p1p2...pn - 2a) mod p2, ..., (p1p2...pn - 2a) mod pn > ⋮ < (p1p2...pn - a) mod p1, (p1p2...pn - a) mod p2, ..., (p1p2...pn - a) mod pn >
carry = 0 --, +-	-a + 1 ⋮ -2 -1	< (p1p2...pn - a + 1) mod p1, (p1p2...pn - a + 1) mod p2, ..., (p1p2...pn - a + 1) mod pn > ⋮ < (p1p2...pn - 2) mod p1, (p1p2...pn - 2) mod p2, ..., (p1p2...pn - 2) mod pn > < (p1p2...pn - 1) mod p1, (p1p2...pn - 1) mod p2, ..., (p1p2...pn - 1) mod pn >

Figure 2: Disjoint digit sets based on condition $4a + 1 \leq p_1p_2...p_n$.

The algorithm for a disjoint set SDNR/RNS digit adder is executed accordingly (adapted from Kuczborski, 1993):

IN PARALLEL FOR $1 \leq index \leq n$ **DO**
BEGIN
uncorrected_sum_p$_{[index]}$ = (operand1_p$_{[index]}$ + operand2_p$_{[index]}$) mod p$_{[index]}$
END
carry_out$_i$ = f (uncorrected_sum_p$_1$, uncorrected_sum_p$_2$, ..., uncorrected_sum_p$_n$)
IN PARALLEL FOR $1 \leq index \leq n$ **DO**
BEGIN
corrected_sum_p$_{[index]}$ = f (carry_out$_i$, uncorrected_sum_p$_{[index]}$)
END
IN PARALLEL FOR $1 \leq index \leq n$ **DO**
BEGIN
sum_p$_{[index]}$ = (corrected_sum_p$_{[index]}$ + carry_in$_i$) mod p$_{[index]}$
END

An explanation of parallel algorithms is given in Kung (1988). The algorithm highlights Kung's (1988) first design criteria, which is maximum parallelism (discussed in section 3.2.3.1-Maximum parallelism). As RNS coding is used at the digit level, the disjoint set algorithm exhibits high parallelism, in comparison to the algorithm for a SDNR digit adder. Figure 17 shows the logic block realisation of the disjoint set algorithm.

Kuczborski (1993) states that the proper choice of a digit set guarantees that carry values are not propagated by more than a single position. The inherent parallel execution allows the addition operation to be performed independent of the word lengths of the operands.

3.1.7.1.3 Nondisjoint digit sets

SDNR/RNS addition using nondisjoint digit sets reduces the required dynamic range to represent a number (Kuczborski, 1993). This has several positive characteristics, including the ability to represent a larger digit set using RNS moduli sets which are fewer and smaller moduli. There are four types, or cases, of nondisjoint digit sets. The first case of nondisjoint digit sets is defined by the following condition:

$3a < p_1p_2...p_n < 4a + 1$

Figure 3 shows a diagrammatical representation for the first case of nondisjoint digit sets. Note that some RNS codes represent two digits instead of one. An algorithm will be presented later which can resolve this discrepancy.

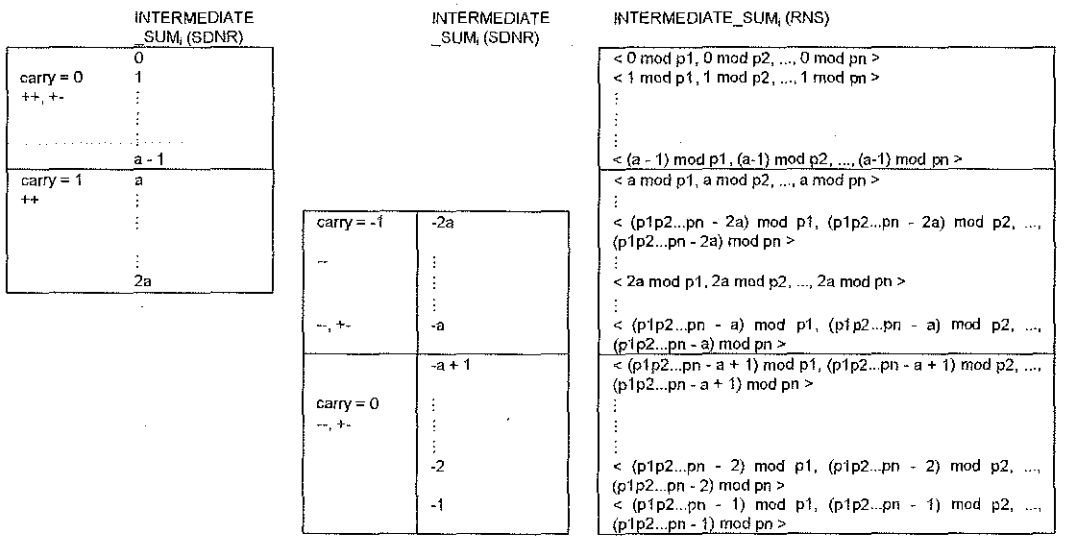

Figure 3: Nondisjoint digit sets based on condition $3a < p_1p_2...p_n < 4a + 1$.

Case two for nondisjoint sets is illustrated in Figure 4. For a SDNR/RNS configuration to qualify for case two, the following condition must be satisfied:

$3a = p_1p_2...p_n$

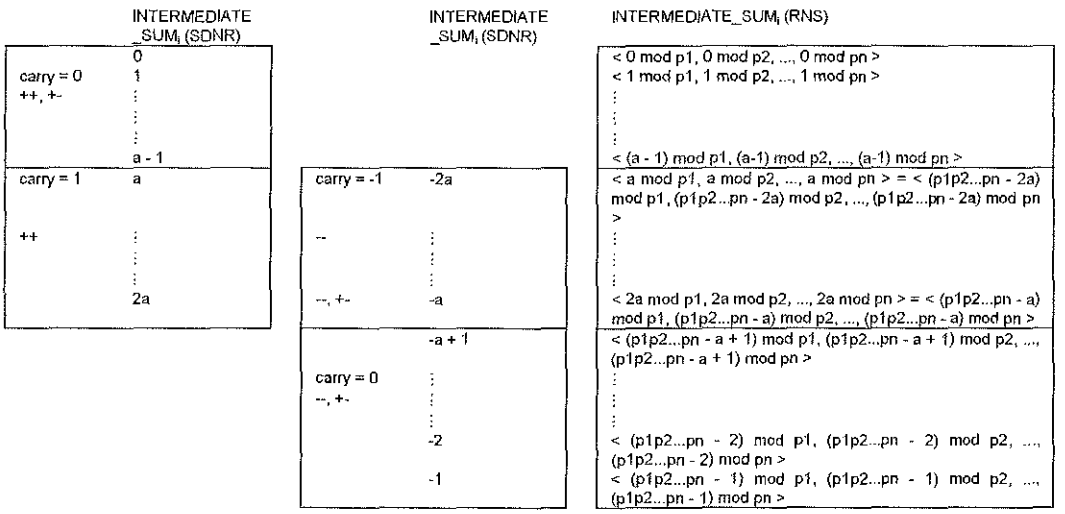

Figure 4: Nondisjoint digit sets based on condition $3a = p_1p_2...p_n$.

Case three for disjoint digit sets is shown in Figure 5. The condition for this case is as follows:

$2a + 1 < p_1p_2...p_n < 3a$

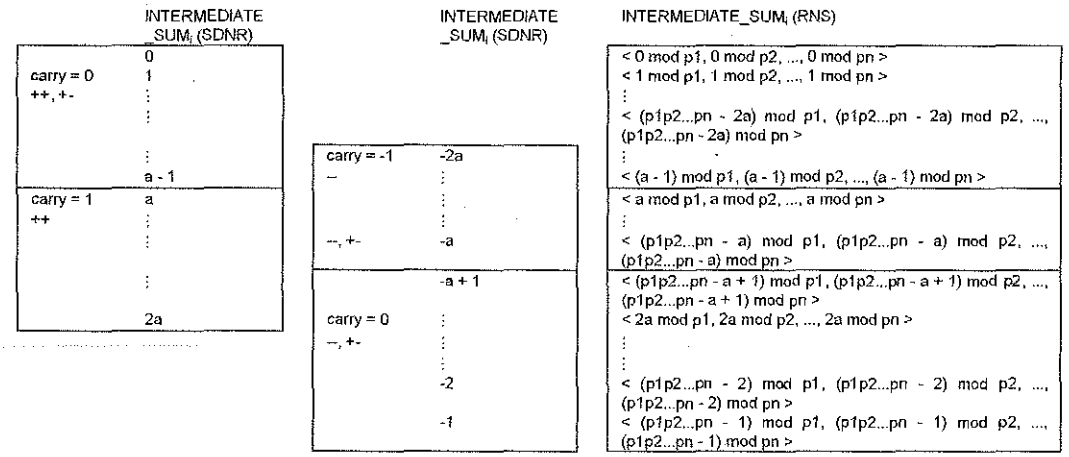

Figure 5: Nondisjoint digit sets based on condition $2a + 1 < p_1p_2...p_n < 3a$.

The fourth case for disjoint sets is described graphically in Figure 6. The condition for this case is as follows:

$2a + 1 = p_1p_2...p_n$

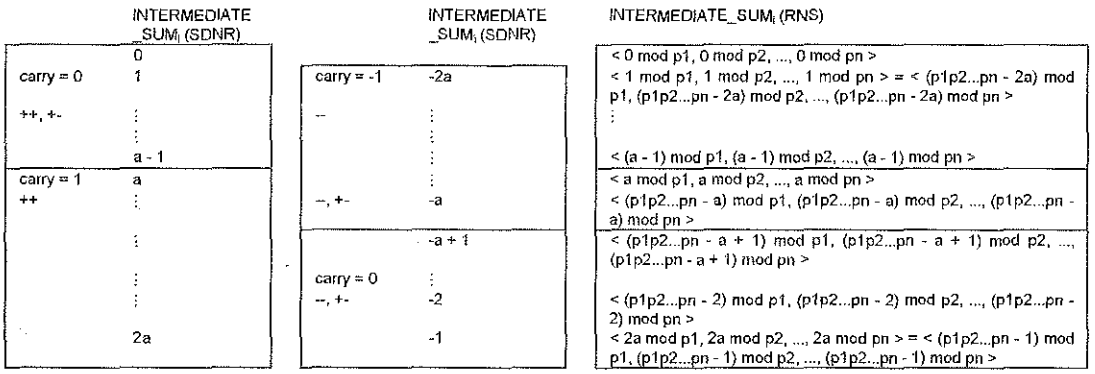

Figure 6: Nondisjoint digit sets based on condition $2a + 1 = p_1p_2...p_n$.

The algorithm for a SDNR/RNS digit adder using disjoint sets can be stated as follows (adapted from Kuczborski, 1993):

IN PARALLEL DO
BEGIN
  IN PARALLEL FOR $1 \le \text{index} \le n$ **DO**
  BEGIN
    $\text{uncorrected\_sum\_p}_{[\text{index}]} = (\text{operand1\_p}_{[\text{index}]} + \text{operand2\_p}_{[\text{index}]}) \bmod p_{[\text{index}]}$
  END
  $\text{operand1\_sign} = f(\text{operand1\_p}_1, \text{operand1\_p}_2, \ldots, \text{operand1\_p}_n)$
  $\text{operand2\_sign} = f(\text{operand2\_p}_1, \text{operand2\_p}_2, \ldots, \text{operand2\_p}_n)$
END
$\text{region} = f(\text{uncorrected\_sum\_p}_1, \text{uncorrected\_sum\_p}_2, \ldots, \text{uncorrected\_sum\_p}_n)$
$\text{carry\_out}_i = f(\text{operand1\_sign}, \text{operand2\_sign}, \text{region})$
IN PARALLEL FOR $1 \le \text{index} \le n$ **DO**
BEGIN
  $\text{corrected\_sum\_p}_{[\text{index}]} = f(\text{carry\_out}, \text{uncorrected\_sum\_p}_{[\text{index}]})$
END
IN PARALLEL FOR $1 \le \text{index} \le n$ **DO**
BEGIN
  $\text{sum\_p}_{[\text{index}]} = (\text{corrected\_sum\_p}_{[\text{index}]} + \text{carry\_in}_i) \bmod p_{[\text{index}]}$
END

The nondisjoint digit-adder algorithm is still parallel by nature, but there are more inherent stages, in comparison to the disjoint case, which must be processed in a serial manner. Therefore,

Kung's (1988) first design criteria (refer to section 3.2.3.1-Maximum parallelism for more information), favours the disjoint digit adder. Figure 18 shows the logic block realisation of the nondisjoint set SDNR/RNS digit adder algorithm.

Both the disjoint digit-adder and the nondisjoint digit-adder have the potential to achieve maximum pipelinablility (Kung's second design criteria; refer to section 3.2.3.2-Maximum pipelinability for more information). Both algorithms imply predictable data dependencies, regularity, and local connections, all of which play a major part in increasing concurrency and pipelining. Kung's (1988) fourth design criteria states that regular communication should be encouraged. Both adder algorithms use local and static communication. These factors contribute towards regular communication. The third and fifth design criteria described in Kung (1988) can be achieved by choosing the optimal values for the radix, moduli, and the digit set.

During the analysis and design phases of the project, the set theory of arithmetic decomposition was used to verify the structure and operation of the chosen SDNR/RNS digit adder configuration. In the next section, the set theory of arithmetic decomposition is explained.

3.1.7.1.4 Set theory of arithmetic decomposition

Carter and Robertson (1990) state that the set theory of arithmetic decomposition is a method for designing complex addition/subtraction circuits at any radix using strictly positional, sign-local number systems. With the theory, the design of circuits to implement the addition is reduced to applying a set of rewrite rules to an equation involving set addition and set scalar multiplication of digit sets that represent the inputs and outputs of the adder.

3.1.7.1.4.1 Definitions

Carter and Robertson (1990) defines a strictly positional number representation as one which the value of a number, whether positive or negative, is computed by a single formula. Furthermore, in sign-local representations, the sign digit does not affect the value of any other digit in the number. As the SDNR satisfies both of these criteria, the set theory of arithmetic decomposition can be used to design and verify a SDNR/RNS digit adder array.

According to Carter and Robertson (1990), a digit set is characterised by two parameters:

1. Diminished cardinality (δ). This parameter is equal to the number of elements in a digit set minus one.
2. Offset (ω). This parameter is the magnitude of the smallest element.

A digit set is denoted as $<\delta^{\omega}>$. Using the concepts of diminished cardinality and offset, a digit set D is defined as follows (Carter and Robertson, 1990):

A digit set D is a sequence of $\delta + 1$ consecutive integers, $\{-\omega + 0, ..., -\omega + \delta\}$.
$\delta \geq 1$. At radix r, $\delta \leq (2r - 2)$.
$\delta \geq \omega \geq 0$ which implies that $0 \in D$.
$\delta \geq r - 1$ which implies that $r \geq 2$.

Carter and Robertson (1990) also specify auxiliary definitions. The following auxiliary definition can be applied to SDNR/RNS representation:

If $\delta > (r - 1)$, then the digit set is redundant.

When using the set theory of arithmetic decomposition it is possible to perform two operations on sets of integers:

1. Set addition.
2. Set scalar multiplication.

Given sets of integers D_i, set addition is defined as follows:

$$D_0 + D_1 = \{(d_0 + d_1) \mid d_0 \in D_0 \text{ and } d_1 \in D_1\}$$

Based on integer addition, set addition is both associative and commutative (Carter and Robertson, 1990). Given, in addition to D_i, set addition is defined as:

$$sD = \{(s \cdot d) \mid s \text{ is an integer and } d \in D\}$$

Based on integer multiplication, set scalar multiplication is associative, commutative, and both right and left distributive over set addition (Carter and Robertson, 1990). Set scalar multiplication takes precedence over set addition.

An arithmetic set expression, according to Carter and Robertson (1990), is a collection of weighted digit sets involving set addition and set scalar multiplication. It is defined as:

$$\sum_{i=0}^{N} s_i D_i$$

where s_i is a scalar.
D_i is a digit set.

An arithmetic set expression that represents a digit set is called a composite digit set and has:

$$\delta_c = \sum_{i=0}^{N} s_i \delta_i$$

and:

$$\omega_c = \sum_{i=0}^{N} s_i \omega_i$$

Therefore, the set expression:

$$8\langle 1^1\rangle + 4\langle 2^0\rangle + 2\langle 1^1\rangle + \langle 2^0\rangle$$

$\delta_c = (8)(1) + (4)(2) + (2)(1) + (2) = 20$
$\omega_c = (8)(1) + (2)(1) = 10$

The resulting composite digit set is $\langle 20^{10}\rangle$.

Carter and Robertson (1990) state that the notion of composite digit sets is of prime importance since it indicates that digit sets of high diminished cardinality can be represented by weighted sums of digit sets of lower diminished cardinality. For example, a four bit two's complement number represents the digit set:

$$\langle 15^8\rangle = \{-8, \ldots, -1, 0, 1, \ldots, 7\}$$

for which the representation as an arithmetic set expression is:

$$8\langle 1^1\rangle + 4\langle 1^0\rangle + 2\langle 1^0\rangle + \langle 1^0\rangle$$

For binary addition and subtraction, all high cardinal digit sets can be represented as weighted sums of binary ($\delta = 1$) and ternary ($\delta = 2$) digit sets (Carter and Robertson, 1990).

Carter and Robertson (1990) also point out that the information content of digit sets and composite digit sets is defined to be the number of distinct signals (or bits) required in the physical realisation. The weighting radix is a number raised to a digit position index by which each successive digit set in a set expression is multiplied. For example, a binary system has a weighting radix of 2. Carter and Robertson (1990) state that the selection of the weighting radix in an arithmetic unit represents a compromise between operational speed and the complexity and cost of design.

3.1.7.1.4.2 Decomposition equations

Carter and Robertson (1990) introduce the decomposition operator ($\Leftarrow$), which indicates that the right-hand arithmetic set expression is to be transformed into the left-hand expression. A decomposition relation has a digit set or composite digit set on both right- and left-hand sides of the decomposition operator. For example, a two digit radix r complement adder can be specified as follows:

$$r^2\langle 1^1\rangle + r\langle (r-1)^0\rangle \Leftarrow (r\langle 1^1\rangle + \langle (r-1)^0\rangle) + (r\langle 1^1\rangle + \langle (r-1)^0\rangle) + \langle 1^0\rangle$$

The final $\langle 1^0\rangle$ digit set on the right-hand side represents the carry in.

The algorithm for an SDNR/RNS digit adder consists of two main stages. The algorithm can be represented as a pair of decomposition equations as follows:

Stage 1: $r\langle 2^1\rangle + \langle 2t^t\rangle \Leftarrow \langle 2a^a\rangle + \langle 2a^a\rangle$

where (from right to left) $\langle 2a^a\rangle$ = operand Y.
$\langle 2a^a\rangle$ = operand X.
$\langle 2t^t\rangle$ = corrected intermediate sum.
$\langle 2^1\rangle$ = carry out.
r = weighted radix.

Stage 2: $\langle 2a^a\rangle \Leftarrow \langle 2t^t\rangle + \langle 2^1\rangle$

where (from right to left) $\langle 2^1\rangle$ = carry in.
$\langle 2t^t\rangle$ = corrected intermediate sum.
$\langle 2a^a\rangle$ = final sum.

and for the a value:

$$r - 1 \geq a \geq \text{FLOOR}\left(\frac{r}{2}, 1\right) + 1$$

for the threshold value t:

$$1 \leq r - a \leq t \leq a - 1$$

The second SDNR/RNS arithmetic operation which was analysed during the project was multiplication. A description of SDNR/RNS multiplication proceeds this section.

3.1.7.2 SDNR/RNS multiplication

The SDNR/RNS data representation allows parallel addition, subtraction, and magnitude comparisons. However, the issue of multiplication was examined more closely to see if it was a feasible SDNR/RNS arithmetic operation.

One of the characteristics of an SDNR/RNS arithmetic system is it's ability to handle very large numbers. For instance, a conventional 64-bit integer has relatively large magnitude, and it would be

expected that the SDNR/RNS arithmetic processor be able to handle such a number efficiently during operations such as addition, or subtraction. What if, however, a multiplication involving two conventional 64-bit integers, which could potentially result in a 128-bit number, was required? Multiplication in the arithmetic system could be performed in a number of ways, including methods such as multiplying by conventional notation, by the RNS, or even by SDNR/RNS.

3.1.7.2.1 Conventional multiplication

When multiplying, two operands are required. One operand is called the multiplier and the other is called the multiplicand (Waser and Flynn, 1982). Example 3.1.7.2.1.1 illustrates multiplication.

3.1.7.2.1.1 Example

Multiplicand				1	1	0		6
Multiplier	*			1	0	1	*	5
				1	1	0		$(6 * 2^0)$
Partial products			0	0	0			$(0 * 2^1)$
		1	1	0				$(6 * 2^2)$
Final product		1	1	1	1	0		30

***

For the conventional and SDNR/RNS methods, the following generalised procedure is executed for multiplication (Waser and Flynn, 1982):

1. First, calculate partial products, then
2. calculate sums of partial products to obtain result.

For both the conventional and SDNR/RNS methods, the second stage of multiplication can be completed using SDNR/RNS adders. Partial product generation using the conventional method can be achieved by using matrix generation and reduction techniques. That is, a modified version of Booth's algorithm can be used to generate the partial products. Booth's algorithm and its derived modification are discussed in many books, including Waser and Flynn (1982), and Kung (1988). The modified version of Booth's algorithm, based on 2-bit encoding, can be characterised as follows:

1. The multiplier must be encoded into groups of 3 bits.
2. For two's complement multiplication, the complement of the multiplicand must be calculated.
3. Number of partial products generated = n/2; where n = maximum length of multiplier or multiplicand (bits).
4. Number of multiplication processing elements required for parallel processing = n/2; where n = maximum length of multiplier or multiplicand (bits).
5. Number of adding stages required to sum partial products = $\log_2 n$; where n = maximum length of multiplier or multiplicand (bits).

The modified version of Booth's algorithm is widely used for multiplication because of these characteristics.

3.1.7.2.2 RNS multiplication

Multiplication can be performed more efficiently by using the RNS. A disadvantage of the RNS scheme is that extremely large numbers can not be handled very easily, because of the dynamic range restriction.

The largest unsigned 8-bit operand equals $2^8 - 1 = 255_{10}$. If two 8-bit operands are multiplied together, the largest possible result is $(2^8 - 1)^2 = 255 * 255 = 65\ 025$. Therefore, the RNS dynamic range must be at least 65 025. Taking the case of multiplying two 64-bit operands dictates the RNS dynamic range to be at least $(2^{64} - 1)^2 = 3.40 * 10^{38}$! This is an extremely large dynamic range. Possible moduli sets can be determined from either one of the following popular guidelines listed in Abdallah and Skavantzos (1995) (refer to section 3.1.5.1-Selection of moduli for a list of the guidelines).

Guideline three seems to be the only suitable choice for such a large dynamic range. However, to find relatively prime moduli for the dynamic range $3.40 * 10^{38}$ hardly seems feasible. Even if a moduli set could be found, the moduli themselves would be so large that the ROM lookup tables required would be too big and too slow. In spite of this setback, RNS is still very efficient at multiplication. Therefore, is it possible to combine SDNR and RNS to perform multiplication with large operands? As it will become apparent, SDNR/RNS multiplication is possible, but not without problems. A theoretical analysis of SDNR/RNS multiplication will first be discussed, followed by the implementation aspects.

3.1.7.2.3 SDNR/RNS multiplication

3.1.7.2.3.1 Theoretical analysis

The problem of multiplication in the SDNR/RNS number system can be understood by first analysing how addition and subtraction are executed. The algorithm for addition and subtraction in the SDNR/RNS scheme is similar to the SDNR algorithm. For SDNR/RNS addition/subtraction, the algorithm, based on a radix r, is as follows:

1. A symmetric digit set is selected. The digit set takes the form {-a, -a + 1, -a + 2, ..., -1, 0, 1, ..., a - 2, a - 1, a}.
2. An extended symmetric digit set is selected so that all intermediate sums can be represented. The extended digit set takes the form {-2a, -2a -1, -2a - 2, ..., 1, 0, 1, ..., 2a - 2, 2a - 1, 2a}. RNS moduli are chosen so that the extended digit set can be represented.
3. A threshold value (t) is set. The threshold value determines carry propagation values. The threshold value must satisfy $1 \leq r - a \leq t \leq a - 1$ for restricted carry propagation.
4. From the operands, intermediate sums are calculated.
5. Based on the threshold value t, intermediate sum and carry corrections are determined.
6. Based on the corrections, final sums are calculated.

From the addition/subtraction algorithm, it is clear that the boundaries of the extended digit set (-2a and 2a) are selected to ensure that an addition involving -a and -a (which results in -2a), or +a and +a (which results in +2a) can be represented. For multiplication, a similar principle applies. That is, the an extended digit set must exist, so that multiplications involving the largest numbers in the digit set can be accommodated. Thus, given the digit set {-a, ..., 0, ..., a}, an extended digit set must be chosen to satisfy $\{-a^2, ..., 0, ..., a^2\}$. An ideal algorithm for SDNR/RNS multiplication can be given as follows:

1. A symmetric digit set is selected. The digit set takes the form {-a, -a + 1, -a + 2, ..., -1, 0, 1, ..., a - 2, a - 1, a}.
2. An extended symmetric digit set is selected so that all intermediate sums can be represented. The extended digit set takes the form $\{-a^2, -a^2 -1, -a^2 - 2, ..., 1, 0, 1, ..., a^2 - 2, a^2 - 1, a^2\}$. RNS moduli are chosen so that the extended digit set can be represented.
3. A threshold value (t) is set. The threshold value determines carry propagation values. The threshold value must satisfy $1 \leq r - a \leq t \leq a - 1$ for restricted carry propagation.
4. From the operands, partial products are calculated.
5. Based on tbe threshold value t and partial products, intermediate sums and carry corrections are determined.
6. Based on the corrections, final sums are calculated.

The problem with the multiplication algorithm is that no procedure exists to convert partial products, which can take any value in the digit set $\{-a^2, ..., 0, ..., a^2\}$, back into the normal digit set specified by {-a, ..., 0, ..., a}, without violating the restricted carry principle for SDNR. In the

addition/subtraction algorithm, carries are restricted to -1, 0, or 1. This allows for parallel addition/subtraction. For multiplication, carries are not restricted. This disallows parallel multiplication. The following example highlights this point.

3.1.7.2.3.1.1 Example

For simplicity, this example is performed in SDNR.

r = 10
a = r/2 + 1
= 10/2 + 1
= 6
t = a - 1
= 6 - 1
= 5

Addition

Digit set = {-6, -5, -4, ..., -1, 0, 1, ..., 4, 5, 6}
Extended digit set = {-12, -11, -10, ..., -1, 0, 1, ..., 10, 11, 12}

		5	6	
+		4	6	
		(9)	(12)	Intermediate sum
		$(\bar{1}\ )$	(2)	Corrected sum
(1)		(1)		Carries
(1)		(0)	(2)	Final sum

From the example, SDNR/RNS addition is relatively straight forward.

Multiplication

Digit set = {-6, -5, -4, ..., -1, 0, 1, ..., 4, 5, 6}
Extended digit set = {-36, -35, -34, ..., 1, 0, 1, ..., 34, 35, 36}

		5	6	
*		4	6	
		(30)	(36)	Partial product
	(20)	(24)	(0)	Partial product
?	?	?	?	Final sum

This is as far as the multiplication algorithm will go without violating the restricted carry rule of SDNR. Intermediate corrections cannot be performed as the resulting intermediate multiplication values cannot be represented in the normal digit set without violating the restricted carry rule.

***

Therefore, is multiplication in the SDNR/RNS number system possible? The answer to this question is yes, but under very severe restrictions. A digit set must be devised which can support SDNR/RNS multiplication, while maintaining the restricted carry set {-1, 0, 1}. The digit set must satisfy the extended digit sets for addition {-2a, ..., 0, ..., 2a} and multiplication $\{-a^2, ..., 0, ..., a^2\}$. If equivalent extended digits sets for addition and multiplication can be derived, then restricted carry propagation can be guaranteed. This results in the following equation being simplified to obtain the boundary values for the normal digit set {-a, ..., 0, ..., a}:

$a^2 = 2a$
$a = 2$

Thus, if a = 2, then parallel addition/subtraction and multiplication is possible. To find the radix that is compatible with this a value, the following redundancy equations are required:

For minimum redundancy:

$a = r/2 + 1$

$r = 2(a - 1)$
$= 2(2 - 1)$
$= 2$ (this radix is invalid in SDNR, as the condition $r > 2$ must be satisfied).

For maximum redundancy:

$a = r - 1$

$r = a + 1$
$= 2 + 1$
$= 3$

Taking the case for maximum redundancy, for a = 2, the radix is 3. Therefore, for parallel addition/subtraction and multiplication, the radix is restricted to 3. However, multiplication is simplified when using the digit set {-2, -1, 0, 1, 2} because the operations then become a series of selected left shifts and additions. For instance, when multiplying by 2, a left shift is required, when multiplying by 1, no shift is required, and when multiplying by 0, no multiplication is required (the result is zero).

When dealing with digital circuits, radix 3 is not favourable, because direct conversion from binary to this radix requires a radix conversion algorithm. In comparison, a radix which has a base of 2 can be directly converted to that radix by grouping the bits in the number. For example, a binary number can be converted to a radix 4 ($= 2^2$) number by grouping bits by twos. Therefore, a pseudo-radix 4 SDNR digit set will be devised, which would allow easy binary conversion, and simplified parallel multiplication.

For a normal SDNR radix 4 number, the digit set for minimum and maximum redundancy is {-3, -2, -1, 0, 1, 2, 3}. However, by using carrier sense logic, the digit set can be reduced to the threshold digit set {-2, -1, 0, 1, 2}. The carrier sense logic algorithm is as follows:

IF $(X_n > t)$ **OR** $(X_n \geq t)$ **AND** $(X_{n-1} \geq t)$ **THEN**
$X_n = X_n - r$
ELSEIF $(X_n < t)$ **OR** $(X_n \leq t)$ **AND** $(X_{n-1} \leq t)$ **THEN**
$X_n = X_n + r$
ELSE
$X_n = X_n$
ENDIF

The next example illustrates SDNR/RNS multiplication.

3.1.7.2.3.1.2 Example

For simplicity, this example is performed in SDNR.

$r = 4$
$a = t$
$= 2$

Digit set = {-2, -1, 0, 1, 2}
Extended digit set = {-4, -3, -2, -1, 0, 1, 2, 3, 4}

Multiplication:

$56_{10} * 46_{10} = 2\ 576_{10}$

$56_{10} = 320_4$
$46_{10} = 232_4$

Convert to SDNR radix 4 representation:

	3	2	0
	$(\bar{1})$	(2)	(0)
(1)			
(1)	$(\bar{1})$	(2)	(0)

	2	3	2
	$(\bar{2})$	$(\bar{1})$	(2)
(1)	(1)		
(1)	$(\bar{1})$	$(\bar{1})$	(2)

Partial products

			(1)	$(\bar{1})$	(2)	(0)	
*			(1)	$(\bar{1})$	$(\bar{1})$	(2)	
			(2)	$(\bar{2})$	(4)	(0)	Partial product #1
		$(\bar{1})$	(1)	$(\bar{2})$	(0)	(0)	Partial product #2
	$(\bar{1})$	(1)	$(\bar{2})$	(0)	(0)	(0)	Partial product #3
(1)	$(\bar{1})$	(2)	(0)	(0)	(0)	(0)	Partial product #4

Additions

r = 4
a = r - 1
= 4 - 1
= 3
t = a - 1
= 3 - 1
= 2

Digit set = {-3, -2, -1, 0, 1, 2, 3}
Extended digit set = {-6, -5, ..., -1, 0, 1, ..., 5, 6}

Add partial products #1 and #2:

		(2)	$(\bar{1})$	(0)	(0)	
+	$(\bar{1})$	(1)	$(\bar{2})$	(0)	(0)	
	$(\bar{1})$	(3)	$(\bar{3})$	(0)	(0)	Sum
	$(\bar{1})$	$(\bar{1})$	(1)	(0)	(0)	Corrected sum/carry sense
	(1)	$(\bar{1})$				Carries
	(0)	$(\bar{2})$	(1)	(0)	(0)	Final sum

Add partial products #3 and #4:

		$(\bar{1})$	(1)	$(\bar{2})$	(0)	(0)	(0)	
+	(1)	$(\bar{1})$	(2)	(0)	(0)	(0)	(0)	
	(1)	$(\bar{2})$	(3)	$(\bar{2})$	(0)	(0)	(0)	Sum
	(1)	$(\bar{2})$	$(\bar{1})$	$(\bar{2})$	(0)	(0)	(0)	Corrected sum/carry sense
		(1)						Carries
	(1)	$(\bar{1})$	$(\bar{1})$	$(\bar{2})$	(0)	(0)	(0)	Final sum

Add sums of partial products #1 and #2, and #3 and #4:

			(0)	$(\bar{2})$	(1)	(0)	(0)	
+	(1)	$(\bar{1})$	$(\bar{1})$	$(\bar{2})$	(0)	(0)	(0)	
	(1)	$(\bar{1})$	$(\bar{1})$	$(\bar{4})$	(1)	(0)	(0)	Sum
	(1)	$(\bar{1})$	$(\bar{1})$	(0)	(1)	(0)	(0)	Corrected sum/carry sense
			$(\bar{1})$					Carries
	(1)	$(\bar{1})$	$(\bar{2})$	(0)	(1)	(0)	(0)	Final sum

Convert to conventional radix 4 representation:

	(1)	(0)	(0)	(0)	(1)	(0)	(0)
-		(1)	(2)	(0)	(0)	(0)	(0)
		(2)	(2)	(0)	(1)	(0)	(0)

$220100_4 = 2\,576_{10}$

***

3.1.7.2.3.2 Practical analysis

In theory, SDNR/RNS multiplication is possible. The target technology for this system is CMOS. A hypothetical implementation would take the form of a systolic array, with each processing element containing at least one SDNR/RNS partial product generator (multiplier) and one SDNR/RNS adder. The purpose of the adder component would be to sum the result from that processing element's partial product generator with the current digit sum calculated from the previous processing element. Kung (1982) and Kung (1988) discuss systolic arrays in detail.

For implementation, a choice in moduli is required to represent the extended digit set {-4, -3, -2, -1, 0, 1, 2, 3, 4} at the digit level. The required dynamic range for this set is 9. The smallest moduli set which would satisfy this dynamic range is (3, 4). This moduli set is characterised by a dynamic range of 12. Therefore, a disjoint SDNR/RNS digit set can be used. Table 4 lists the digit set and corresponding RNS representation.

Table 4: SDNR/RNS digit set.

SDNR digit	RNS number	
	(SDNR digit) MOD p1	(SDNR digit) MOD p2
-4	2	0
-3	0	1
-2	1	2
-1	2	3
0	0	0
1	1	1
2	2	2
3	0	3
4	1	0

The SDNR/RNS configuration presented in Table 4 is not practically feasible. First, four bits would be required to represent each digit in SDNR/RNS form. Given that two bits are required to represent each grouping of bits in conventional notation, the redundancy factor for this particular SDNR/RNS configuration would be 100%. To put this into perspective, a 128-bit SDNR/RNS number would be required to represent a conventional 64-bit integer. The redundancy factor in this case is unacceptable, as it is an inefficient way to store a number and makes it impractical to implement such a circuit. The fact that low radices are not well represented in SDNR/RNS data representation is noted in Kuczborski (1993).

Second, there are severe restrictions on the digit set. The only digit set that accommodates SDNR/RNS multiplication is {-2, -1, 0, 1, 2}. One advantage of the digit set {-2, -1, 0, 1, 2} is that it suits digital circuits because multiplication becomes a simple series of selected left shifts and additions. Other advantages of SDNR/RNS multiplication are that it inherits characteristics such as modularity, regularity, and computational fault isolation.

The disadvantages present in the scheme stem from the fact that the system lacks basic flexibility. For example, with an SDNR/RNS adder, a designer can freely choose the radix, the moduli set, and the redundancy factor. In contrast, the SDNR/RNS multiplier can only be realised if a certain configuration is used. This configuration is detailed in Table 5.

Table 5: Configuration for a SDNR/RNS multiplication scheme.

Radix	p1	p2	Dynamic range	a	Number of digits required to represent a conventional 64-bit integer	Number of bits required to represent a conventional 64-bit integer
4	3	4	12	3	32	128

Despite the inflexibilities, the proposed SDNR/RNS multiplication method has similar characteristics to the modified version of Booth's algorithm, and they are as follows:

1. The multiplier and multiplicand must be encoded into SDNR/RNS notation.
2. Two's complement multiplication is performed relatively easily because of the SDNR component in the SDNR/RNS notation.
3. Number of partial products generated = n/2; where n = maximum length of multiplier or multiplicand (bits).
4. Number of multiplication processing elements required for parallel processing = $(n/2)^2$; where n = maximum length of multiplier or multiplicand (bits).
5. Number of adding stages required to sum partial products = $\log_2 n$; where n = maximum length of multiplier or multiplicand (bits).

The major difference between the modified version of Booth's algorithm and the SDNR/RNS methods is the number of multiplication processing elements required for pure parallel multiplication. The SDNR/RNS method requires a lot more of these processing elements ($(n/2)^2$) than the modified

version of Booth's algorithm (n/2). Furthermore, the modified version of Booth's algorithm allows larger bit encoding schemes, which in turn reduces the number of resultant partial products. This, in turn, reduces the number of partial product processing elements required.

It has been shown that SNDR/RNS multiplication is possible. However, the gains expected from using such a scheme are outweighed by its inflexibilities. One of the main inflexibilities is that the extended digit set must satisfy $\{-a^2, \ldots, 0, \ldots, a^2\}$. The only digit set able to comply with this criteria is $\{-a, \ldots, -1, 0, 1, \ldots, +a\} = \{-2, -1, 0, 1, 2\}$. If a technique is devised to overcome this digit set restriction, so that larger and more efficient radices and digit sets can be accommodated, then SDNR/RNS multiplication may become practically feasible.

3.2 Computer architecture

This section describes the major characteristics of VLSI in order to justify its application to this project.

3.2.1 VLSI characteristics

According to Hwang and Briggs (1984), the key attributes of VLSI computing structures are simplicity and regularity, concurrency and communication, and computation intensiveness.

3.2.1.1 Simplicity and regularity

VLSI chips comprise of hundreds of thousands of identical components. To cope with that complexity, simple and regular designs are essential. Hwang and Briggs (1984) state that VLSI systems based on simple, regular layout are likely to be modular and adjustable to various performance levels.

Hwang and Briggs (1984) associate the simplicity and regularity of VLSI designs to the issue of cost. The issue of cost effectiveness has always been a major concern in designing special purpose VLSI systems. Specifically, their cost must be low enough to excuse their limited applicability. Special purpose design costs can be reduced by the use of appropriate simple and regular architectures (Hwang and Briggs, 1984).

The digit adder that was implemented was simple and regular. Characteristics of the SDNR/RNS number system made sure that the digit adder was modular, and was adjustable to various performance levels.

3.2.1.2 Concurrency and communication

Hwang and Briggs (1984) highlight the fact that the degree of concurrency in a VLSI computing structure is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing.

Coordination and communication become significant when a large number of PEs working simultaneously (Hwang and Briggs, 1984). This is especially true for VLSI technology where routing costs dominate power, time, and area required to implement a computation. Therefore, algorithms need to be designed that support high degrees of concurrency, while employing only simple, regular communication and control. Hwang and Briggs (1984) point out that the locality of interprocessor communications is a desired property to have in any processor arrays.

First, the digit adder algorithm lends itself to parallelism. Second, the SDNR/RNS number system restricts carry propagation to one position. Therefore, a SDNR/RNS digit adder is only required to communicate with its closest neighbours. Both of these characteristics promote concurrency and simple communications.

3.2.1.3 Computation intensiveness

Hwang and Briggs (1984) state that VLSI processing structures are suitable for implementing compute-bound algorithms rather than I/O-bound computations. In a compute-bound algorithm, the number of computing operations is larger than the total number of input or output elements. Otherwise, the problem is I/O bound (Hwang and Briggs, 1984). For example, Kung (1982) states that the ordinary matrix-matrix multiplication algorithm represents a compute bound task, since every entry in a matrix is multiplied by all entries in some row or column of the other matrix. In comparison, adding two matrices is I/O bound, since the total number of adds is not larger than the total number of entries in the two matrices.

The I/O bound problems are not suitable for VLSI because VLSI packaging must be constrained with limited I/O pins. Therefore, a VLSI device must balance internal computation with the I/O bandwidth. Having knowledge of the I/O imposed performance limit helps prevent overkill in the design of special purpose VLSI processors (Hwang and Briggs, 1984).

The SDNR/RNS digit adder algorithm exhibits I/O bound behaviour. However, the data could be input and output in a serial manner, which would balance internal processing with I/O bandwidth.

Hwang and Briggs (1984) assert that the choice of an appropriate architecture for any electronic system, including VLSI, is very closely related to the implementation technology. In VLSI, the constraints of power dissipation, I/O pin count, communication delays, difficulty in design and layout, and so on are less critical in other technologies. Conversely, VLSI offers fast and inexpensive computational elements.

Properly designed parallel structures that need to communicate only with their nearest neighbour gain the most from VLSI, according to Hwang and Briggs (1984). Valuable time is lost when modules that are far apart must communicate.

3.2.2 Clock distribution schemes

According to Kung (1988), a system wide clock signal controls the activities in a large synchronous system. The purpose of the clock signal is twofold. First, the clock acts as a sequence reference, and second, it acts as a time reference (Kung, 1988):

1. Sequence reference: The clock transitions serve the purpose of defining successive instants at which system state changes may occur.
2. Time reference: The period between clock transitions accounts for wiring and element delays in paths from the output to input of clocked elements.

Kung (1988) asserts that clock distribution is a critical issue for systolic arrays. This is because the clock signal dictates the activities of the entire system. In view of this fact, clock distribution problems must be overcome in the design of the array processor (or an array of digit adders). The main problems are from clock skew (Kung, 1988). That is, each digit adder, or PE, in an array may not receive the clock signal at the same time. Reasons for the clock skew can be attributed to the different path lengths from the clock generator to each PE, or other reasons, such as process variations for different clock paths.

To overcome clock path problems, Kung (1988) suggests designing the array processor with an H-tree clock distribution scheme. This scheme can be used to distribute the clock signal to regular arrays, such that every PE has the same distance from the clock generator. H-tree layouts are shown in Figure 7, Figure 8, and Figure 9 for various types of arrays.

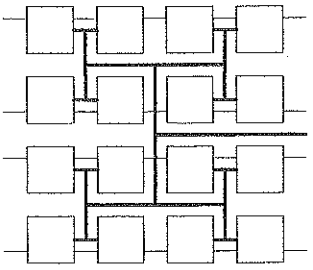

Figure 7: H-tree layout for clocking a linear array.

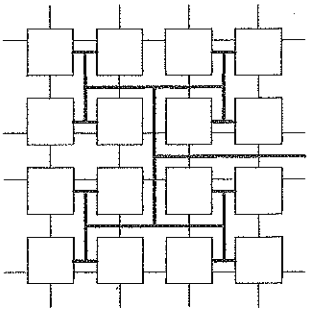

Figure 8: H-tree layout for clocking a square array.

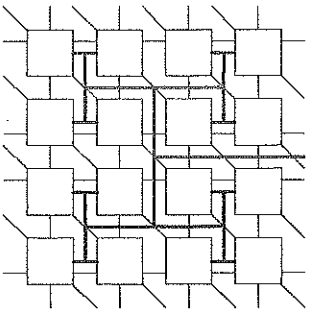

Figure 9: H-tree layout for clocking a hexagonal array.

Kung (1988) states that even though clock path problems can be overcome, clock skew can not be completely resolved. It has been shown that an arbitrarily large linear systolic array can be synchronised by a global clock by the use of pipelined clocks (Kung, 1988). However, an attempt to synchronise a 2-D array usually encounters a clock skew proportional to the size of the array. Clock skew can only be overcome by employing asynchronous design principles in the design of digital systems. Architectures known as wavefront arrays employ such principles. By using wavefront arrays, the clocking problem can be alleviated, because only correct sequencing, and not timing, is required for correct operation (Kung, 1988).

3.2.3 VLSI array algorithms

A traditional measure of VLSI circuit efficiency involves determining the area-time complexity of a circuit. Area-time complexity measures depend on two factors, computation time (T) and circuit area (A). Kung (1988) states that the complexity measure AT^2 is very popular in lower-bound analysis of VLSI algorithms. However, Kung (1988) points out that an AT^2 measure seems to offer little practical implication in VLSI system design. A practical measure f (A, T) depends strongly on individual applications. For example, if speed is more important, then more weighting must be placed on the time parameter T. On the other hand, if cost is more important, then more emphasis needs to be placed upon the area parameter A. Kung (1988) finds that little relationship has been established

between the special measure AT^2, and a practical measure f (A, T). It is on this observation that Kung (1988) has defined new design criteria for array algorithms.

The new criterion for measuring the efficiency of a VLSI array realisations include computation, communication, memory, and input/output (I/O) aspects. These criterion will be used in determining the optimal configuration for an SDNR/RNS digit adder design.

3.2.3.1 Maximum parallelism

Two algorithms with equivalent performance in a sequential computer may perform quite differently in parallel processing environments. Kung (1988) states that an algorithm will be favoured if it expresses a higher parallelism, which is exploitable by the computing arrays.

3.2.3.2 Maximum pipelinability

Most signal processing algorithms demand very high throughput rates and are computationally intensive, in comparison to their I/O requirements. The use of pipelining is often very natural in regular and locally interconnected networks. Kung (1988) states that, as a result, a major part of concurrency in array processing will be derived from pipelining. To maximise the throughput rate, the best algorithm must be used. Unpredictable data dependency may severely jeopardise the processing efficiency of a highly regular and structured array algorithm. Effective VLSI arrays are inherently highly pipelined and hence require well structured algorithms with predictable data movements. Iterative methods with dynamic branching, dependent on data produced during the process, are less well suited for pipelined architectures.

3.2.3.3 Balance among computations, communications, and memory

Kung (1988) states that a good array algorithm should offer a sound balance between different bandwidths incurred in different communication hierarchies to avoid data draining or unnecessary bottlenecks. Balancing the computations and various communication bandwidths is critical to the effectiveness of array computing. In today's technology, it is not hard to improve the computation bandwidth. However, as Kung (1988) points out, it is much harder to increase the I/O bandwidth. In this case, the pipeline techniques are especially suitable for balancing computation and I/O because the data tend to engage as many processors as possible before leaving the array. This will reduce I/O bandwidth for outside communication. For certain computation bound problems, such as matrix multiplication, Fast Fourier Transforms and sorting, if the computation bandwidth is increased while the I/O bandwidth is kept constant, the size of local memory has to increase in order to balance the computation with I/O.

3.2.3.4 Trade off between computation and communication

To make the interconnection network practical, efficient, and affordable, regular communication should be encouraged. According to Kung (1988), major issues affecting the communication regularity include local versus global, static versus dynamic, and data-independent versus data-dependent interconnection modules. The criterion should maximise the trade-off between interconnection cost and throughput. To conform with the communication constraints imposed by VLSI, a lot of emphasis has recently been placed on local and recursive algorithms. Take Discrete Fourier Transforms, witb a computation cost of $O(N^2)$, and Fast Fourier Transform computing, having an associated cost of $O(N\log_2 N)$. The FFT, in terms of computation, is favoured by almost one order of magnitude. On the other hand, the DFT is characterised by simple communication needs because is belongs to a locally recursive class. The FFT computation requires a global interconnection. Kung (1988) highlights that this leads to a contrasting trade-off. For example, an algorithm requiring only a static network is preferable to one requiring a dynamic network, since a static interconnection network is physically easier to construct.

3.2.3.5 Numerical performance and quantisation effects

Kung (1988) states that numerical behaviour depends on many factors, such as the word length of the computer and the algorithms used. Often, additional computations may be wisely utilised to improve the overall numerical performance. However, the tradeoff between computation and numerical behaviour is very problem dependent, and there is no general rule to apply. For example, an FFT computation is computationally cost effective, and at the same time numerically well behaved. An exception to there being no general rule is that extra computation can always be used to increase the word length, and thus assures improved performance.

3.3 Implementation technology

There are two target technologies for the VLSI arithmetic circuits. The first is Complementary Metal Oxide Semiconductor, and the second is Gallium Arsenide. CMOS was used to implement the chosen SDNR/RNS digit adder.

3.3.1 Complementary metal oxide semiconductor

As the name implies, complementary MOS technology employs MOS transistors of both n- and p-type polarities. Information on CMOS can be found in many books, including Glasser and Dobberpuhl (1988), Weste and Eshraghian (1994), and Pucknell and Eshraghian (1994). Weste and Eshraghian (1994) describe CMOS by listing its main attributes:

1. Power supply. The power supply can range from 1.5 to 15V.
2. Power dissipation. Static power dissipation is almost zero. Power is dissipated during logic transitions.
3. Fully restored logic levels. That is, output settles at the supply voltage V_{DD}, or ground, V_{SS}.
4. Precharging characteristics. Both n- and p-type devices are available for precharging a bus to V_{DD} and V_{SS}.
5. Transition times. Rise and fall times are of the same order.
6. Packing density. Logic circuits can be implemented in dense structures.
7. Layout. CMOS encourages regular and easily automated layout styles.

Clocking rates in CMOS are relatively slow. CMOS can only support clocking rates in the order of megahertz. Therefore, the clock skew problem outlined in section 3.2.2-Clock distribution schemes is not as big a problem as it is for GaAs, which has the ability to operate at gigahertz frequencies.

3.3.1.1 CMOS technology

The semiconductor silicon forms the basic starting material for a wide variety of integrated circuits. A Metal Oxide Semiconductor (MOS) structure is created by superimposing several layers of conducting, insulating, and transistor forming materials to create a sandwich-like structure. Weste and Eshraghian (1994) state that these structures are created by a series of chemical processing steps involving oxidation of the silicon, diffusion of impurities into the silicon to give it certain conduction characteristics, and deposition and etching of aluminium on the silicon to provide interconnection in the same way that a printed circuit board is constructed. This construction process is carried out on a single crystal of silicon, which is available in the form of thin, flat circular wafers around 15cm in diameter (Weste and Eshraghian, 1994). CMOS technology provides two types of transistors, an n-type transistor (nMOS) and a p-type transistor (pMOS). These are fabricated in silicon by using either negatively diffused (doped) silicon that is rich in electrons, or positively doped silicon that is rich in holes. After the fabrication steps, a typical MOS structure includes distinct layers called diffusion

(silicon which has been doped), polysilicon (crystalline silicon used for interconnection), and aluminium, separated by insulating layers (Weste and Eshraghian, 1994).

For the n-transistor, the structure consists of a section of p-type silicon (called the substrate) separating two areas of n-type silicon. This structure is constructed by using a chemical process that changes selected areas in the positive substrate into negative regions rich in electrons. The area separating the n regions is capped with a sandwich consisting of silicon dioxide (an insulator), and a conducting electrode (usually polycrystalline silicon-poly) called the gate. Similarly, for the p-transistor the structure consists of a section of n-type silicon separating two p-type areas. In common with the n-transistor, the p-transistor also has a gate electrode. The transistors also have two additional connections, designated the source and drain, these being formed by the n (p in the case of a p-device) diffused regions. The gate is a control input. It affects the flow of electrical current between the source and the drain. A connection to the substrate forms the fourth terminal of an MOS transistor.

The four main CMOS technologies are:

1. n-well process.
2. p-well process.
3. Twin-tub process.
4. Silicon on insulator.

For the implementation stage of this project, a p-well fabrication process was assumed. Weste and Eshraghian (1994) state that p-well processes are preferred in circumstances where the characteristics of n- and p-transistors are required to be more balanced than that achievable in an n-well process. The p-well process has better p-transistors than an n-well process because the transistor that resides in the native substrate tends to have better characteristics. Due to that fact that p-transistors have lower gain than their n counterparts, the n-well process increases this difference, while a p-well process moderates the difference. The reason that lower gain is experienced by pMOS devices in CMOS is because electron mobility in silicon is much greater than hole mobility. Table 6 (adapted from Streetman, 1990) lists the mobilities for comparison.

Table 6: Electron and hole mobilities in silicon.

μ_n (cm^2/V-s)	μ_p (cm^2/V-s)
1350	480

3.3.1.2 Layout design rules

Weste and Eshraghian (1994) state that layout rules, also referred to as design rules, can be considered as a prescription for preparing the masks used in fabrication of integrated circuits. The rules provide a necessary communication link between circuit designer and process engineer during the manufacture phase. The main objective associated with layout rules is to obtain a circuit with optimum yield (functional circuits versus non functional circuits) in as small an area as possible without compremising reliability of the circuit (Weste and Eshraghain, 1994).

The design rules primarility address two issues:

1. The geometrical reproduction of features that can be reproduced by the mask-making and lithographical process.
2. The interactions between different layers.

There are several approaches that can be taken in describing the design rules. These include micron rules stated at some micron resolution, and lambda (λ)-based rules. Micron design rules are usually given as a list of minimum feature sizes and spacings for all the masks required in a given process. For example, the minimum active width might be specified as 1μm. According to Weste and Eshraghian (1994), this is the normal style for industry. The lambda-based design rules are based on a single parameter, λ, which characterises the linear feature, that is, the resolution of the complete wafer

implementation process, and permits first-order scaling. Pucknell and Eshraghian (1994) state that lambda-based rules lead to a simple set of rules for the designer, and wide acceptance of the rules by a large cross-section of the fabrication houses and silicon brokers. Furthermore, the scaling feature of the lamba-based rules may help to give designs a longer lifetime. However, Weste and Eshraghian (1994) report that while these rules have been successfully used for 1.2 - 4μm processes, they will probably not suffice for submicron processes.

The CAD system used during the implementation stages of the project incorporated a scalable CMOS (SCMOS) design rule checker. In other words, the implemented circuits were based on lambda-based rules.

3.3.1.3 Latchup

Pucknell and Eshraghian (1994) state that a problem which is inherent in the p-well and n-well processes is due to the relatively large number of junctions which are formed in these structures and the consequent presence of parasitic transistors and diodes. Latchup is a condition in which the parasitic components give rise to the establishment of low-resistance conducting paths between the power rail (V_{DD}) and ground rail (V_{SS}) with disastrous results. Latchup may be induced by glitches on the supply rails or by incident radiation. Weste and Eshraghian (1994) and Pucknell and Eshraghian (1994) describe the condition of latchup in more detail.

Latchup may be prevented in two basic ways:

1. Latchup resistant CMOS processes.
2. Layout techniques.

The first prevention method was outside the scope of this project. Weste and Eshraghian (1994) detail processes which are latchup resistant.

In this project, layout techniques were used to minimise any possibility of latchup taking place. Weste and Eshraghian (1994) point out that the key technique to reduce latchup is to make good use of substrate and well contacts. In most current processes, the possibility of latchup occurring in internal circuitry has been reduced to the point where a designer need not worry about the effect as long as liberal substrate contacts are used. A few rules were followed in this project which reduced the possibility of internal latchup to a very small likelihood (the rules are listed from Weste and Eshraghian, 1994):

1. Every well must have a substrate contact of the appropriate type.
2. Every substrate contact should be connected to metal directly to a supply pad.
3. Place substrate contacts as close as possible to the source connection of transistors connected to the supply rails (that is, V_{SS} for n-transistors, V_{DD} for p-transistors). A very conservative rule would place one substrate contact for every supply (V_{SS} or V_{DD}) connection.
4. Otherwise a less conservative rule is place a substrate contact for every 5 - 10 transistors, or every 25 - 100μm.
5. Lay out n- and p-transistors with a packing of n-transistors toward V_{SS} and packing of p-transistors toward V_{DD}. Avoid complicated structures that intertwine n- and p-transistors in checkerboard styles.

3.3.1.4 Power dissipation

There are three components that establish the amount of power dissipated in a CMOS circuit, and they are:

1. Static dissipation due to leakage current or other current drawn continuously from the power supply.

2. Dynamic dissipation due to:
    - Switching transient current.
    - Charging and discharging of load capacitances.
3. Short circuit dissipation.

The static power dissipation is the product of the device leakage current and the supply voltage. Weste and Eshraghian (1994) state that a useful estimation is to allow a leakage current of 0.1nA to 0.5nA per device at room temperature. Then total static power dissipation is obtained as follows:

$$P_S = \sum_{1}^{n} I_l V_{DD}$$

where P_s = static power dissipation.
n = number of transistors.
I_l = leakage current.
V_{DD} = supply voltage.

For a more rounded estimate the following equation can be used (Pucknell and Eshraghian, 1994):

$$P_s = nI_lV_{DD}$$

Pucknell and Eshraghian (1994) state that the dynamic power dissipation is due to energy supplied to charge and discharge the capacitances associated with each switching circuit. Assuming that that output capacitance of a stage can be combined with the input capacitance(s) of the stage(s) it is driving and then represented collectively as C_L, then, for n identical circuits switched by a square wave at frequency f:

$$P_d = C_L V_{DD}^2 f$$

where P_d = dynamic power dissipation.
C_L = load capacitances.
V_{DD} = supply voltage.
f = frequency of square wave (for example, the clock).

Manually determining the load capacitances for each transistor in a large circuit is not very easy. Specifically, the load capacitance seen by a gate is dependent on:

1. The size of the transistors in the gate (self loading).
2. The size and number of transistors to which the gate is connected.
3. The routing capacitance between a gate and the ones it drives.

Weste and Eshraghian (1994) point out that during the transition from either 0 to 1, or 1 to 0, both n- and p-transistors are on for a short period of time. This results in a short circuit pulse from V_{DD} to V_{SS}. The short circuit power dissipation is given by:

$$P_{sc} = I_{mean} V_{DD}$$

where P_{sc} = short circuit power dissipation.
I_{mean} = average current used during logic state transition.
V_{DD} = supply voltage.

Finally, the total power dissipation can be obtained from the sum of the three dissipation components:

$$P_T = P_s + P_d + P_{sc}$$

where P_T = total power dissipation.
P_s = static power dissipation.
P_d = dynamic power dissipation.
P_{sc} = short circuit power dissipation.

The rule, according to Weste and Eshraghian (1994), is to add all capacitances operating at a particular frequency, and then the power should be calculated. The power from other groups operating at different frequencies may be summed afterwards. For a quicker estimation of power, the dynamic power dissipation may be used to estimate total power consumption of a circuit. This is because the dynamic power dissipation is usually the dominant term.

In this project, however, the implemented digit adder contained more than 1 000 transistors. It was impractical to calculate the power dissipation in a detailed manner. Instead, a switch level simulator was going to be used (the software was not available at the time) which had the ability to be modifed to sum the total capacitance switched by each switch on each node over the course of a simulation run. After the simulation run, the total number of clock cycles that would have been simulatated could have been used in conjunction with the capacitance as follows (Weste and Eshraghian, 1994):

$$P_d = \frac{C_T V_{DD}^2}{n_c t_p}$$

where P_d = dynamic power dissipation.
C_T = total switched capacitance.
V_{DD} = supply voltage.
n_c = total number of cycles.
t_p = period of switching frequency.

According to Weste and Eshraghian (1994), there are several ways to minimise power dissipation in a CMOS circuit. DC power dissipation may be reduced to leakage by only using complementary logic gates. The leakage in turn is proportional to the area of diffusion, so the use of minimum-sized transistors is of advantage. Dynamic power dissipation may be limited by reducing supply voltage, switched capacitance, and the frequency at which the logic is clocked.

In the implementation stage of this project, dynamic logic was used. As a result, DC power dissipation was not as minimal as would have been desired. Minimum transistor sizes were used where possible, so leakage current was reduced. Some transistors were resized to improve circuit delays.

3.3.1.5 Fan-in and fan-out

All logic gates have two attributes in common, and they are fan-in and fan-out. Weste and Eshraghian (1994) define the fan-in of a logic gate as the number of inputs the gate has in the logic path beign exercised. For example, a 2-input NOR gate has a fan-in of 2. Conversely, the fan-out of a logic gate is the total number of gate inputs that are driven by a gate output. For example, if the output of the 2-input NOR gate was used by 4 other gates, then the fan-out of the NOR gate would be 4.

The fan-in of a gate affects the speed of the gate. Weste and Eshraghian (1994) recommend that when gates with large numbers of inputs have to be implemented, the best speed-performance may be obtained by using gates where the number of series inputs ranges between about 2 and 5.

3.3.1.6 CMOS logic structures

There are several CMOS logic structures to choose from when implementing a system. In some situations, the area taken by a fully complementary static CMOS gate may be greater than that required, the speed may be too slow, or the function may just not be feasible as a purely complementary

structure. In these cases, it is desirable to implement smaller, faster gates at a cost of increased design and operational complexity, and, possibly, decreased operational margin.

In this project, it was found that fully complementary static CMOS logic was not feasible. If such logic was used, the implemented SDNR/RNS digit adder would have consisted of about twice as many transistors as the implemented adder. As a result, the digit adder would have been nearly twice as large. Therefore, the digit adder was constructed using dynamic logic. Before dynamic logic is introduced, however, the traditional static complementary logic will be described.

3.3.1.6.1 CMOS complementary logic

The CMOS complementary gate has two function determining blocks, an n-block and a p-block. There are normally 2n transistors in an n-input gate. Figure 10 shows the general layout for a CMOS complementary gate.

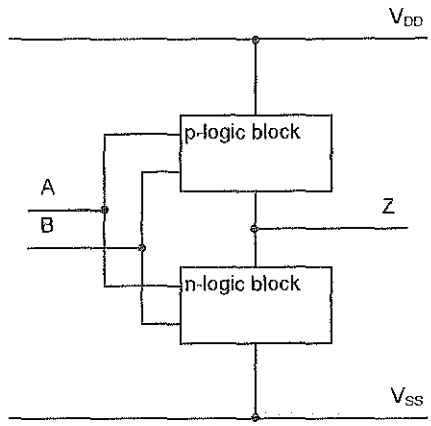

Figure 10: CMOS complementary logic. A and B are arbitrary inputs, and Z is the output.

In general, a CMOS complementary logic gate is formed by using a combination of series- and parallel-transistor (switch) structures. A logic equation can quite easily be converted into a CMOS complementary circuit. The logic equation must be manipulated so that it can be equated in terms of NANDs and NORs. Once this is complete, then the circuit can be constructed by using the following rules:

- n operands in the logic equation being NANDed must be represented by n transistors in series in the n-block, and by n transistors in parallel in the p-block.
- m operands in the logic equation being NORed must be represented by m transistors in parallel in the n-block, and by m transistors in series in the p-block.

By using this complementary form of static logic, logic equations can be implemented with a comparitively high degree of simplicity. However, the circuit area is relatively large because for every one transistor in the n-logic block, there is one transistor in the p-block.

3.3.1.6.2 Dynamic CMOS logic

A basic dynamic CMOS gate is shown in Figure 11. The gate consists of an n-transistor logic structure whose output node is precharged to V_{DD} by a p-transistor and conditionally discharged by an n-transistor connected to V_{SS}. Alternatively, an n-transistor precharged to V_{SS} and a p-transistor discharge to V_{DD} and a p-logic block may be used. The input CLK is a single phase clock. The precharge phase occurs when CLK = 0. The path to the ground is closed via the n-transistor during the evaluate phase, or when CLK = 1.

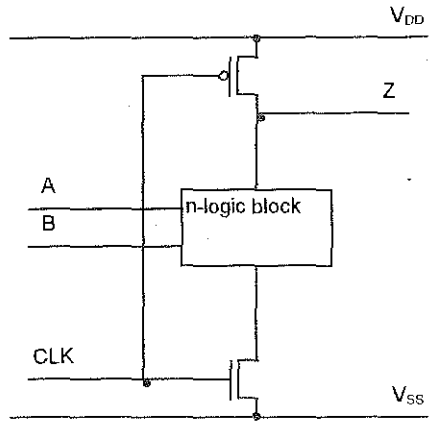

Figure 11: Dynamic CMOS logic. A and B are arbitrary inputs, and Z is the output.

Weste and Eshraghian (1994) state that there are a number of problems associated with dynamic logic. First, the inputs can only change during the precharge phase and must be stable during the evaluation portion of the cycle. If this condition is not met, charge redistribution effects can corrupt the output node voltage. Second, simple single-phase dynamic CMOS gates can not be cascaded. The second problem is very restrictive in creating CMOS circuits. To solve this problem, a special kind of dynamic logic is used instead. It is called domino logic.

3.3.1.6.3 CMOS domino logic

CMOS domino logic incorporates a static CMOS inverter into each logic gate, as shown in Figure 12. Figure 13 reveals the structure of the static inverter (or buffer) used in the logic gate. Weste and Eshraghian (1994) explain the operation of CMOS domino logic in terms of precharing and evaluation. During precharge, the output node of the dynamic gate is precharged high and the output buffer is low. As subsequent logic stages are fed from this buffer, transistors in subsequent logic blocks will be turned off during the precharge phase. When the gate is evaluated, the output will conditionally discharge, allowing the output buffer to conditionally go high. Thus, each gate in sequence can make at most one transition (1 to 0). Hence, the buffer can only make a transition from 0 to 1. In a cascaded set of logic blocks, each state evaluates and causes the next stage to evaluate. In effect, it is like a line of dominos falling down. Any number of logic stages may be cascaded, provided that the sequence can evaluate within the evaluate clock phase. A single clock can be used to precharge and evalute all logic gates within a block.

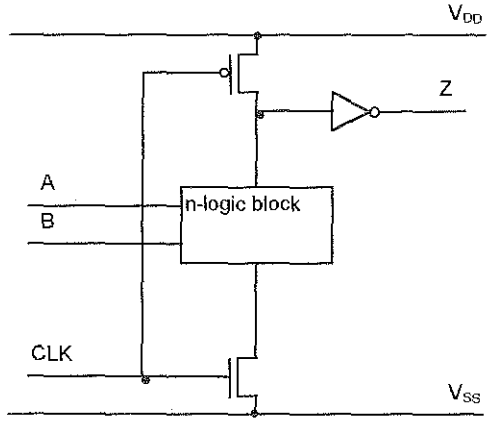

Figure 12: CMOS domino logic. A and B are arbitrary inputs, and Z is the output.

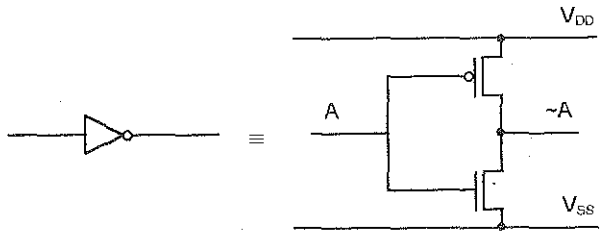

Figure 13: Static CMOS inverter. A is the input, and ~A is the negated output.

Weste and Eshraghian (1994) point out some limitations to this structure. First, each gate must be buffered. However, this may be an advantage. For example, the transistors in the buffer could be resized so that the logic block could effectively drive more gates than if no buffer was used at all. Second, only noninverting structures are possible. Finally, because the logic is still dynamic, charge redistribution can be a problem.

The main reason domino logic was chosen for the implemented SDNR/RNS digit adder was because it reduced the overall number of transistors used, and it allowed gates to be cascaded.

3.3.1.7 Clocking strategies

Clocking strategies were investigated as a part of this project because dynamic logic structures were used during the implementation stage. There are many clocking schemes, ranging from single phase to four phase clocking arrangements.

In the case of single phase clocking, conventional static logic may be used. Furthermore, domino logic may be used to improve speed, and reduce area and dynamic power dissipation. However, according to Weste and Eshraghian (1994), it is difficult to pipeline such logic stages while using a single clock and complement. Two phase clocking strategies make it easier to implement systems where pipelining is desirable. However, for this project, a four phase clocking scheme was used, as this somewhat simplified logic design.

3.3.1.7.1 Four phase clocking

The dynamic logic that has been described has a precharge phase and an evaluate phase. Weste and Eshraghian (1994) state that the addition of a "hold" phase can simplify dynamic logic design. This primarily results from the elimination of charge sharing in the evaluation cycle. Four phase clocking schemes reduce circuit size and increase clocking safeness. Arguments for using such a clocking strategy include the fact that no more clock lines are needed that for two phase clocking if certain four phase structures are used. However, modern designs tend to minimise the number of clock phases used, and employ self-timed circuits to generate special clocks (Weste and Eshraghian, 1994).

The particular four phase clocking scheme used in this project is described in Weste and Estraghian (1994), and is as follows. There are four types of gates characterised by the phase in which evaluation occurs. When using such logic gates, they must be used in the appropriate sequence. The allowable connections between types are illustrated in Figure 14. Figure 15 depicts the four phase clock to be used with the gates shown in Figure 14. Note also that a sample and hold component is used in each gate of Figure 14. This component is called a transmission gate, and its structure is shown in Figure 16.

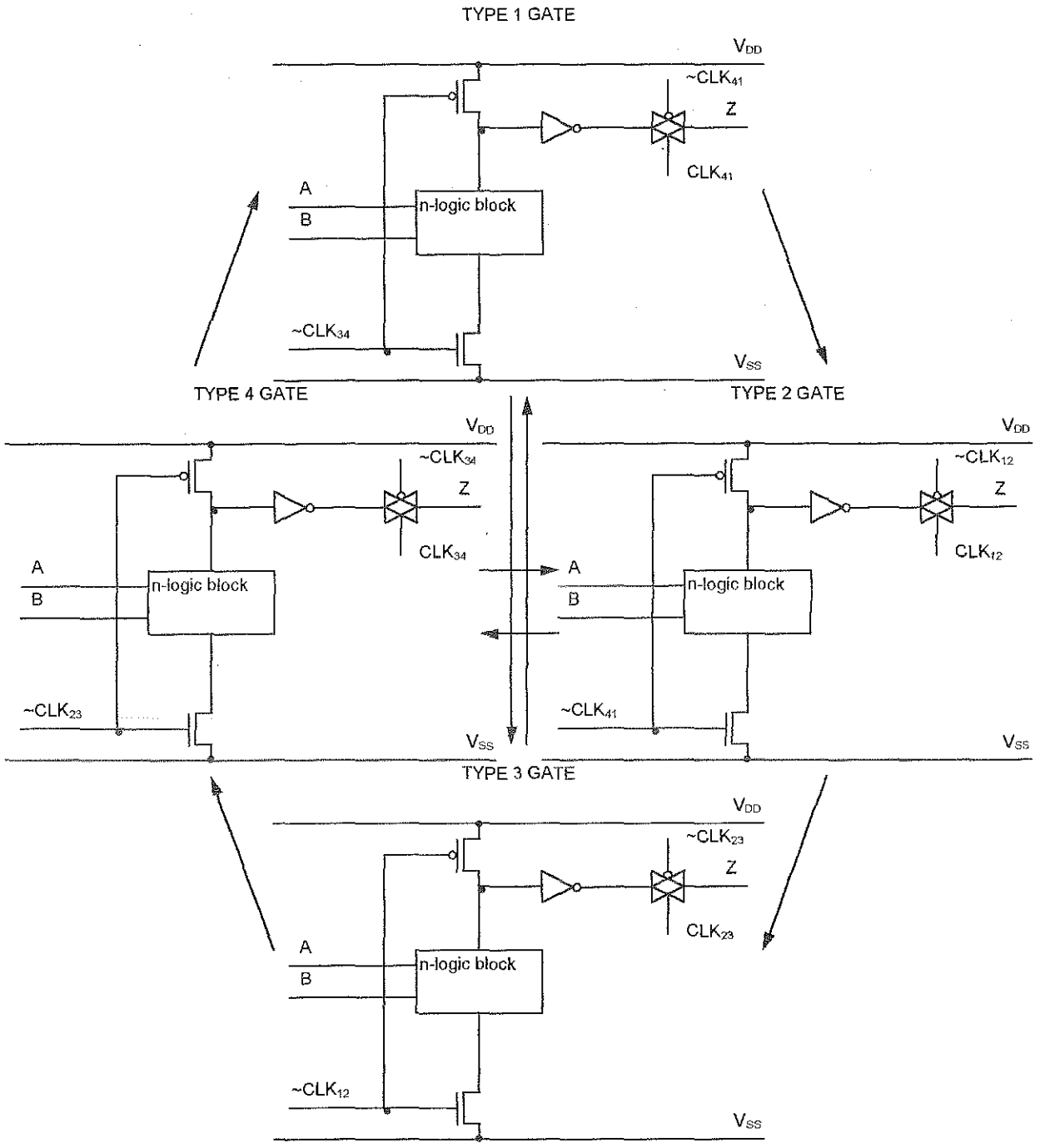

Figure 14: Allowable connections between different gate types for a four phase clocking scheme.

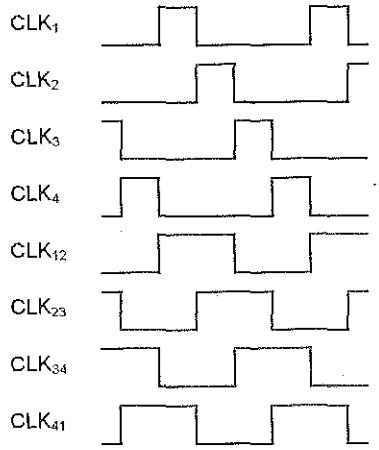

Figure 15: Four phase clock required for the logic gates shown in Figure 14.

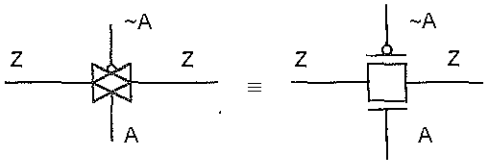

Figure 16: CMOS transmission gate.

By using four phase clocking, four levels of logic may be evaluated per cycle. Alternatively, a two phase logic scheme may be employed by using type 4 gates and type 2 gates, or type 1 gates and type 3 gates.

A problem with four phase clocking is that the clock frequency must be long enough to allows for the slowest gate to evaluate (Weste and Eshraghian, 1994). Thus, fast gates tend to evaluate quickly, and the remainder of the cycle is "dead time". Other system design problems arise when trying to distribute four or more clocks and synchronise them around a large chip.

4. Analysis

This chapter is concerned with identifying and analysing the major characteristics of the SDNR/RNS number system. By choosing a certain configuration based on some guidelines given at the end of the chapter, an optimal design and implementation of an SDNR/RNS arithmetic system is attainable.

4.1 SDNR/RNS configuration analysis

The principle arithmetic circuit in any system is the adder. For this reason, the following SDNR/RNS configuration analysis will be based on choosing a configuration for such a circuit. The other elements of a nonconventional arithmetic system, for example, the conversion circuits, sign detectors, and even multiplication circuits, all depend on the adder, in one way, or another. The three key parameters that are required to create a unique SDNR/RNS configuration are radix, moduli (choice of moduli and how many), and the digit set.

The choice of radix r depends on the desired balance between the increase in storage requirements and the logical complexity of one digit-adder. Avizienis (1961) points out that the relative increase in storage capacity requirements diminishes when r is large. However, when r is large, one digit-adder must accept more values of a digit and the logical circuits become more complex. Ramamoorthy, Potu, and Govind (1988) highlight the obvious advantage in using a radix which is a power of 2. Yang, Lu, and Gilbert (1991) demonstrate that the power of 2 advantage in the implementation costs may be technology independent. Such radices allow easy conversion from the signed-digit to binary, and vice versa. Therefore, a radix is required which will balance storage requirements and logical complexity, and which is a power of 2.

Avizienis (1961) states that minimal-redundancy representations require the least storage capacity for the values of a digit and therefore are preferable to representations with higher redundancy. In addition, he points out that less complicated digit-adder logic may be expected when the least possible number of digit values is employed. On the other hand, maximum-redundant representations allow the fastest and simplest conventional to SDNR/RNS conversion schemes, if the radix chosen is a power of 2. Specifically, when converting between conventional and maximally redundant SDNR/RNS number systems, the second stage in conversion (correcting the converted digit) is not required. For the case of minimally redundant digit sets, the second stage is required. Therefore, if number system conversion is absolutely critical for the given application, then a maximum-redundant digit set should be used. Otherwise, a minimal-redundant digit set should be employed. Avizienis (1961) recommends using the latter kind of digit set, reasons for which are stated previously.

Moduli must be chosen to represent the SDNR/RNS number at the digit level. At this point, a decision must be made as to whether a disjoint or nondisjoint digit set is to be used. The major difference between the two types of digit sets is that nondisjoint sets allow greater numerical range. The disadvantage of having a nondisjoint set is that extra circuitry is required in the digit adder to uniquely identify the operands.

Kuczborski (1993) designed a digit adder for both disjoint digit sets and nondisjoint digit sets. These adders could accept SDNR/RNS digits composed two moduli. More generalised definitions of these adders are shown in Figure 17 and Figure 18. The more generalised cases of digit adders can accept SDNR/RNS digits composed of n moduli (p1, p2, ..., pn). As can be observed, the nondisjoint digit set adder requires four more logic components than does the disjoint digit set version.

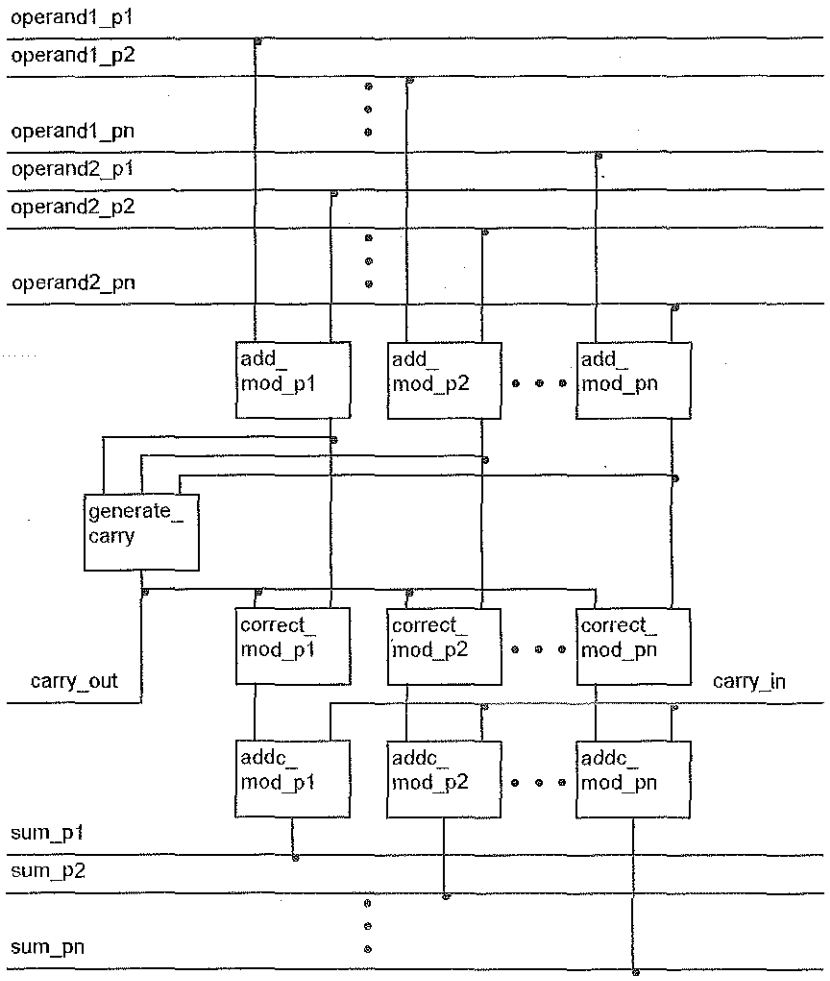

Figure 17: SDNR/RNS disjoint set digit-adder.

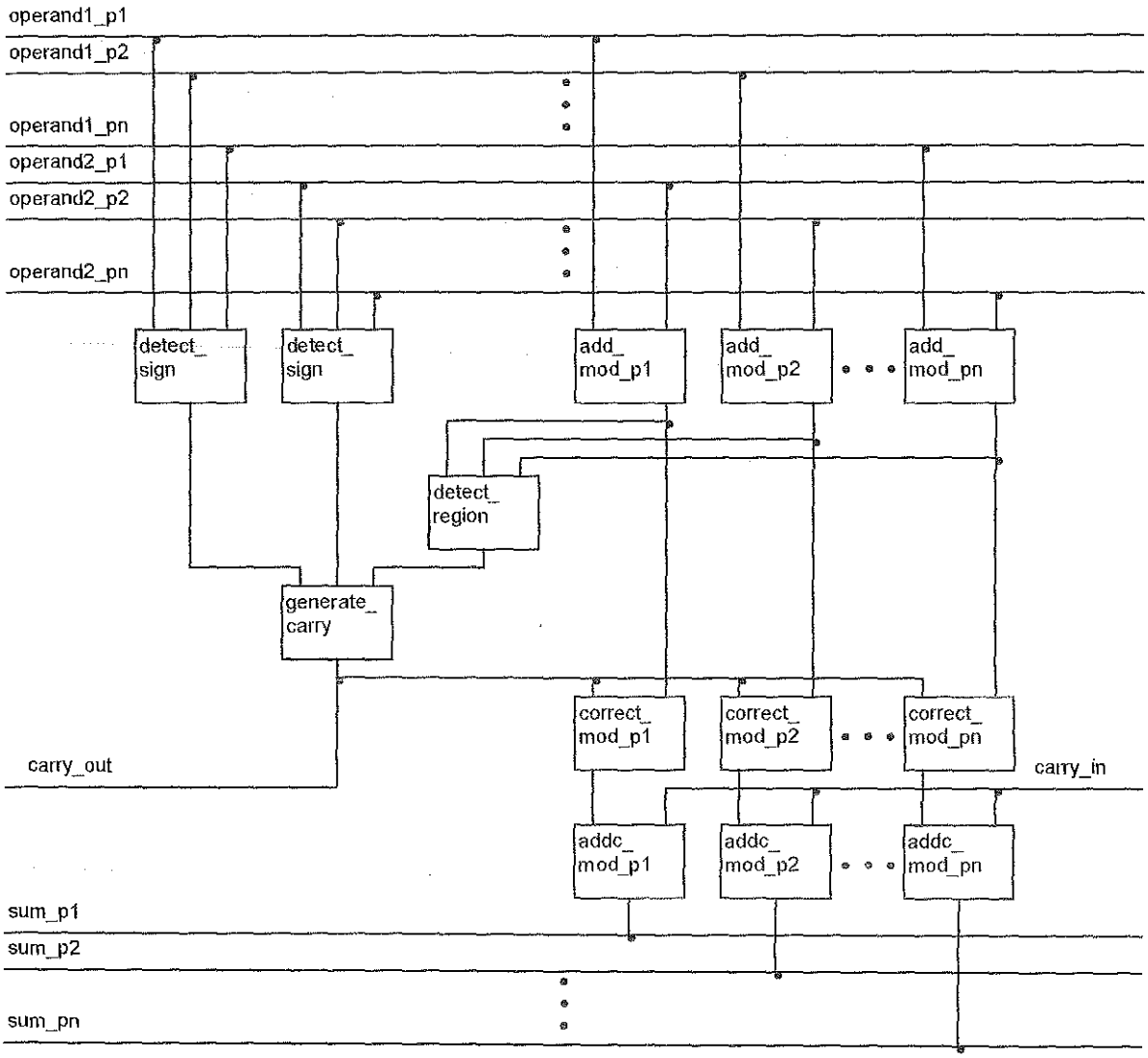

Figure 18: SDNR/RNS nondisjoint set digit-adder.

From Figure 17 and Figure 18, it can be deduced that both adder designs are insensitive, in terms of computational speed, to the number of moduli in the moduli set. That is, the n moduli add operations can be performed in parallel. A limitation to this characteristic is that the length of the generate_carry output transmission lines (carry_out and carry_in) have to be increased as n is increased. As a result, load capacitance on these lines are also increased. This, in turn, increases the propagation delays in the carry_out and carry_in transmission lines. Weste and Eshraghian (1994) suggest using buffers on such lines to decrease propagation times.

If the number of moduli chosen to represent the digit set is large, then this may degrade the speed of the generate_carry and detect_region logic gates in Figure 17 and Figure 18, respectively. This is because as n is increased, so too is the fan-in of the generate_carry and detect_region logic circuits. In turn, this increases the delay in those logic blocks which affects overall adder performance. Kuczborski (1993), however, discusses how complex logic functions, with many inputs, can be decomposed into simpler equations, each with fewer inputs. Logic decomposition can be used at the expense of extra levels of logic. Even though logic decomposition is targeted at Field Programmable Logic Arrays, their application in VLSI is justified as it can be used to increase the speed of the logic. In other words, logic decomposition can be used to reduce the fan-in of the gates to a point where the number of inputs to each logic network is between 2 and 5. Kuczborski, Attikiouzel and Crebbin (1994) present an efficient algorithm for the purpose of decomposing a logic function into simpler components. Luba (1994) describes logic decomposition with some good examples. By using the techniques discussed above, a digit adder can be implemented whose performance is relatively independent from n.

The propagation delay of the modulo addition stages would depend upon the time for the slowest modulo adder to compute a result. The slowest modulus adder will be the one that has to accept the largest range of values, based on statements by Avizienis (1961). For example, if the moduli set (11, 13, 15, 16) was chosen for a particular SDNR/RNS configuration, then the modulo 16 adder, which would exhibit the greatest logic complexity, would require the greatest time to compute results, whereas the modulo 11 adder would require the least time.

Nevertheless, the SDNR/RNS digit adder is very efficient for performing high radix parallel addition. Take, for example, radix 32. An SDNR digit adder would have to contain logic circuits to add two operands of CEILING ($\log_2 32$) = 5 bits each. In comparison, a SDNR/RNS adder with digits coded by an RNS moduli set of (7, 8) would contain a modulo 7 and modulo 8 adder in parallel. In this case, the modulo 8 adder would be slowest, and therefore would dictate the propagation delay for the SDNR/RNS adder, in that an operand can be represented by any one of 8 values, compared to 7 values for the modulo 7 adder. However, the SNDR digit adder must be able to accept 32 digit values for each operand, which would indicate the magnitude of the propagation delay for this type of adder. The SDNR/RNS digit adder must accept two operands of CEILING ($\log_2 8$) + CEILING ($\log_2 7$) = 6 bits each. Therefore, at the cost of an extra bit in storage for each operand, the SDNR/RNS adder exhibits greater efficiency at performing arithmetic at the digit level.

Abhallah and Skavantzos (1995) have developed a list of guidelines for choosing RNS moduli sets. One of the recommendations is that the moduli p_is should be as small as possible, so that operations modulo p_i require minimum computation time. This assertion agrees with Avizienis's (1961) statement regarding the complexity of digit adder logic. Therefore, in choosing a moduli set to satisfy the required digit dynamic range, a tradeoff analysis is required in choosing the smallest possible moduli while maintaining respectable redundancy at the digit level.

In summary, the following guidelines should be adhered to when deciding on the configuration for the SDNR/RNS digit-adder:

Radix guidelines:

- Find a balance between storage requirements and the logic complexity of one digit-adder. In general, minimising storage requirements increases logic complexity. By increasing logic complexity, circuit propagation delay is also increased. However, by careful analysis, storage requirements and logic complexity can both be minimised.
- Choose a radix which is a power of 2.

Digit set guidelines:

- Minimum-redundant digit sets are preferable to digit sets with higher redundancy because they require the least storage capacity for the values of a digit. The logic complexity for minimum-redundant digit sets is small, in comparison to maximum-redundant representations. However, maximum-redundant digit sets allow the fastest and simplest conventional to SDNR/RNS conversion, if the radix is a power of 2. The type of application would indicate what kind of digit set is to be used.
- A nondisjoint digit set increases the range of a SDNR/RNS number at the expense of more logic circuitry. A critical paths analysis indicates that a disjoint digit set adder may minimise the delay in one of the paths. This is an issue open for investigation.

Moduli guidelines (adapted from Abhallah and Skavantzos, 1995):

- Moduli should be relatively prime. That is, there should be no common divisor between any of the moduli in the set (p_1, p_2, ..., p_n).
- The moduli p_is should be as small as possible so that operations modulo p_i require minimum computational time.
- The product of the moduli should be large enough in order to implement the desired dynamic range.

- The moduli p_is should create a balanced decomposition of the dynamic range. That is, the differences between the number of bits to represent the different moduli should not be very large.

4.2 Case studies

An analysis of several optimal SDNR/RNS configurations was required to decide what parameters should be used in the design of the digit-adder. The case studies concentrate on examining the following aspects of a SDNR/RNS digit adder:

1. The radix.
2. The moduli.
3. The digit set (minimum versus maximum redundancy).
4. The number of elements in a RNS moduli set.
5. Optimising memory and delay requirements.

To begin with, the relative merits of maximum and minimum redundant SDNR/RNS digits with RNS moduli sets containing two elements will be discussed. This analysis will be repeated for SDNR/RNS digits with RNS moduli sets consisting of three elements.

For sample data, radices of the power of 2 ranging from 8 to 4096 were analysed. For both minimum and maximum redundant systems, a single configuration was chosen for each radix. The choice was based upon the guidelines outlined in the previous section.

An additional constraint was that all SDNR/RNS configurations selected for analysis had to satisfy the range of a conventional 64-bit unsigned integer. That is, for a particular SDNR/RNS word, the following condition had to be adhered to:

Range (SDNR/RNS word) $\geq 2^{64} - 1$

where $2^{64} - 1 = 1.84 * 10^{19}$

4.2.1 RNS moduli set consisting of two elements

4.2.1.1 Minimum redundancy

Table 7 lists the SDNR/RNS configurations chosen for minimally redundant two moduli digit-adders.

Table 7: SDNR/RNS configurations (2 moduli; minimum redundancy).

Radix	p1	p2	Dynamic range (p1p2)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
8	3	4	12	5	22	88	4	1.84E+19	5.27E+19
16	4	5	20	9	17	85	5	1.84E+19	1.77E+20
32	5	7	35	17	13	78	6	1.84E+19	2.02E+19
64	7	10	70	33	11	77	7	1.84E+19	3.87E+19
128	11	12	132	65	10	80	8	1.84E+19	6.04E+20
256	16	17	272	129	9	81	9	1.84E+19	2.39E+21
512	23	24	552	257	8	80	10	1.84E+19	2.38E+21
1024	32	33	1056	513	7	77	11	1.84E+19	5.92E+20
2048	45	46	2070	1025	6	72	12	1.84E+19	3.69E+19
4096	64	65	4160	2049	6	78	13	1.84E+19	2.36E+21

Figure 19 shows a plot of radix versus the word length. The graph depicts the word storage efficiency for each radix.

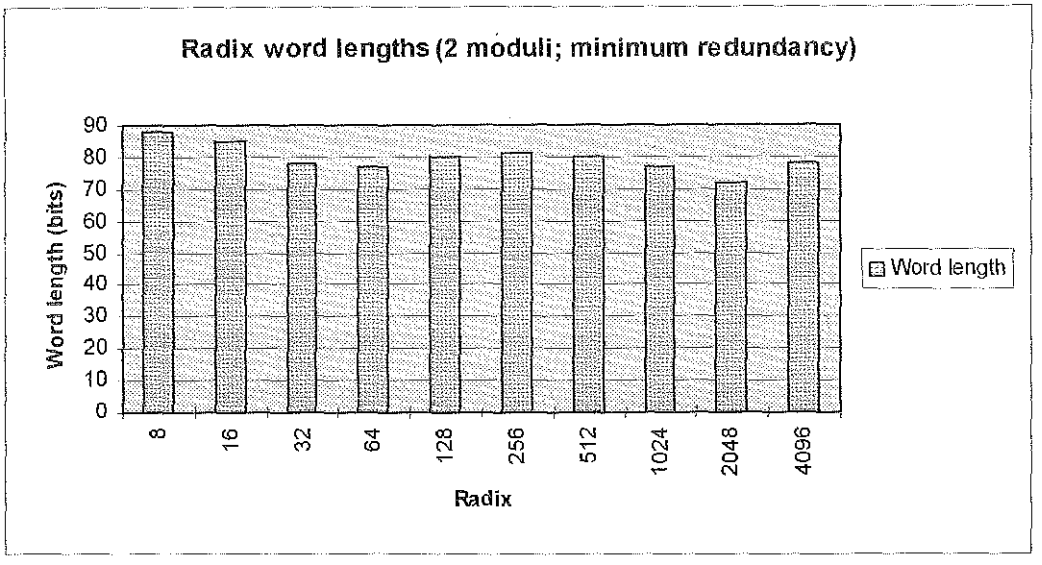

Figure 19: Radix word lengths (2modnli; minimum redundancy).

In Figure 19, notice that the word length assumes a sinusoidal like function. As reported in Kuczborski (1993), the lower radices (in this case, 8 and 16) exhibit excess redundancy, and this translates into longer word lengths.

At radix 256, the word length hits a peak value. The moduli pair for this radix is (16, 17). To represent the first element in the moduli set, CEILING ($\log_2 16$) = 4 bits are required. However, to represent the second element in the moduli set, CEILING ($\log_2 17$) = 5 bits are required. The fifth bit is required to represent the seventeenth value. This means that an extra bit is required at the digit level just so that the second modulus can be properly handled by the adder. Hence, the effect of requiring the extra bit at the digit level is amplified at the word level. In comparison, radices 32, 64, and 2048 seem to exhibit optimal storage requirements for representing a conventional 64-bit integer.

Figure 20 graphically depicts the number of bits required at the digit level for each radix listed in Table 7.

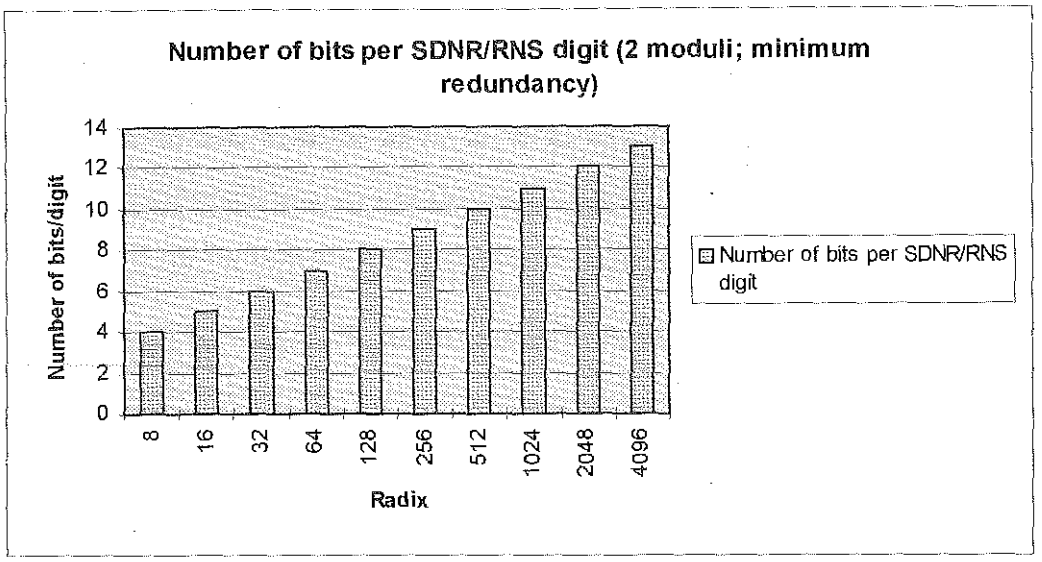

Figure 20: Number of bits per SDNR/RNS digit (2 moduli; minimum redundancy).

Figure 20 indicates the logic complexity for a digit-adder, according to radix. The trend appears to be log-linear. Each successive jump in the number of bits per SDNR/RNS digit can be described mathematically as follows:

$$N = \log_2(r) + 1$$

where N = Number of bits per SDNR/RNS digit.
r = Radix.
= 8, 16, 32, ..., 4096.

As the number of bits required to represent each digit is increased, the greater the logic complexity of the corresponding digit adder. For example, the radix 8 digit-adder needs to be able to handle 4 binary inputs per operand. For this case, logic complexity would be minimal because of the small number of variable inputs (two 4-bit operands). In comparison, the radix 4096 digit-adder needs to be able to accept 13 binary inputs per operand. For the radix 4096 digit-adder, there are 26 variable inputs (two 13-bit operands), which would result in logic which is quite complex. Increased logic complexity is not favourable (Avizienis, 1961).

The radices 32, 64, and 2048 are characterised by minimal word lengths. From these three candidates, radix 32 exhibits the smallest bits required per digit. Thus, the radix 32 configuration is characterised by efficient storage and relatively low logic complexity when representing a conventional 64-bit integer.

4.2.1.2 Maximum redundancy

Table 8 lists the SDNR/RNS configurations chosen for maximally redundant two moduli digit-adders. These maximum redundant representations will be analysed and compared with the corresponding minimum redundant configurations from the previous section.

Table 8: SDNR/RNS configurations (2 moduli; maximum redundancy).

Radix	p1	p2	Dynamic range (p1p2)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
8	3	5	15	7	22	110	5	1.84E+19	7.38E+19
16	5	7	35	15	16	96	6	1.84E+19	1.84E+19
32	7	9	63	31	13	91	7	1.84E+19	3.69E+19
64	10	13	130	63	11	88	8	1.84E+19	7.38E+19
128	15	17	255	127	10	90	9	1.84E+19	1.18E+21
256	23	24	552	255	8	80	10	1.84E+19	1.84E+19
512	32	33	1056	511	8	88	11	1.84E+19	4.72E+21
1024	45	46	2070	1023	7	84	12	1.84E+19	1.18E+21
2048	64	65	4160	2047	6	78	13	1.84E+19	7.38E+19
4096	87	95	8265	4095	6	84	14	1.84E+19	4.72E+21

A graph showing the word lengths for each radix listed in Table 8 is shown in Figure 21. The graph represents the word storage efficiency for each radix.

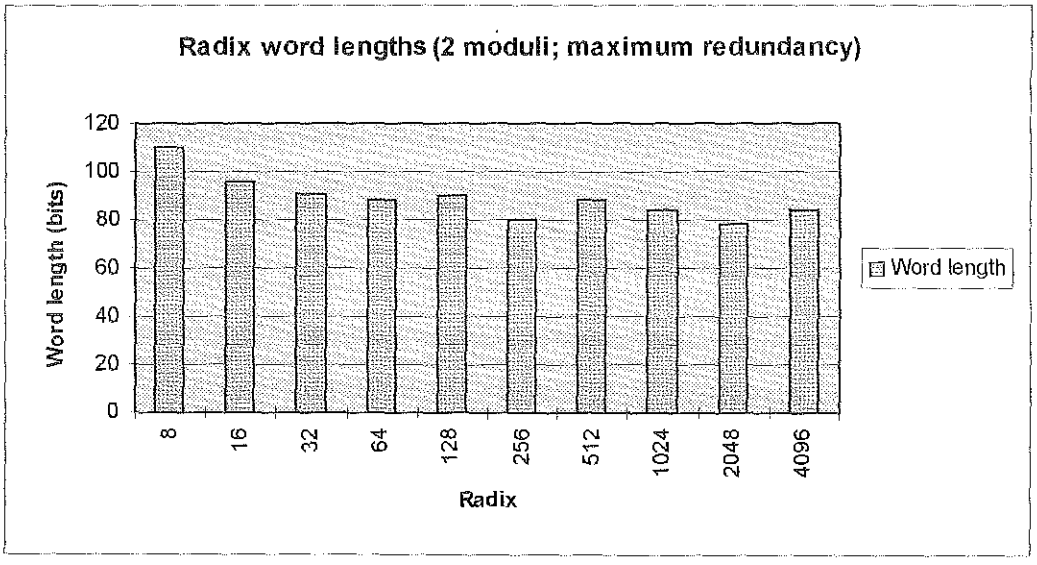

Figure 21: Radix word lengths (2 moduli; maximum redundancy).

Figure 21 exhibits a similar nature to that of Figure 19. That is, a sinusoidal like pattern can be seen from the trend. Like Figure 19, it is evident from Figure 21 that the lower radices are characterised by inefficient storage at the word level. However, in Figure 19, radix 256 is relatively inefficient in being able to store the range required by a conventional 64-bit unsigned integer. According to Figure 21, however, it can be seen that radix 256 can be used with a maximum redundant digit set, while maintaining a relatively small overall word length. For maximum redundant digit sets, radices 256 and 2048 seem to be optimal, from a word level point of view.

The number of bits required to represent the SDNR/RNS configurations listed in Table 8 are graphically illustrated in Figure 22.

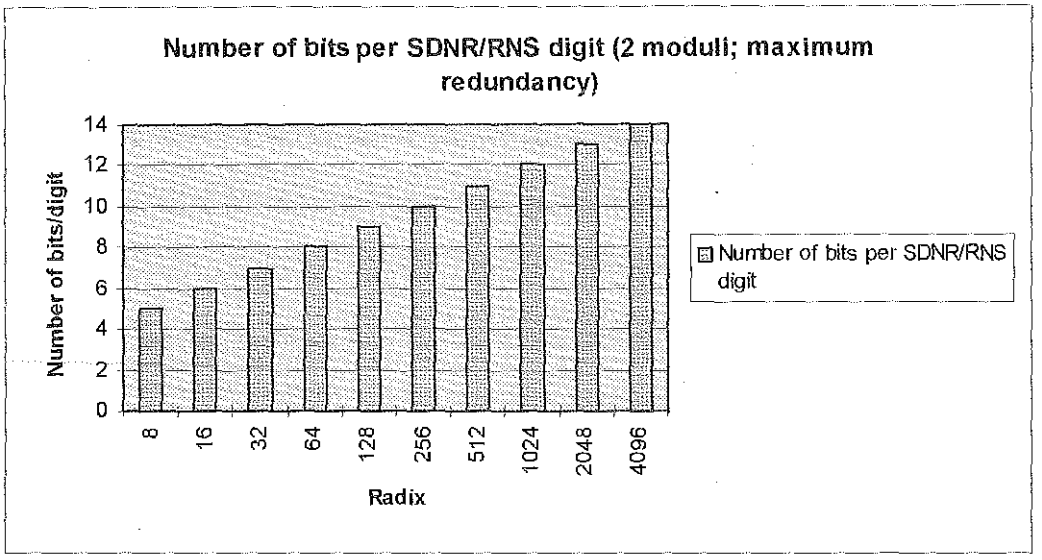

Figure 22: Number of bits per SDNR/RNS digit (2 moduli; maximum redundancy).

Figure 22 indicates the logic complexity for a digit-adder, in terms of radix. The trend, like in Figure 20, seems to be log-linear. Each successive jump in the number of bits per SDNR/RNS digit can be described mathematically as follows:

$$N = \log_2(r) + 2$$

where N = Number of bits per SDNR/RNS digit.
r = Radix.
= 8, 16, 32, ..., 4096.

For maximum redundant configurations, an additional bit is required at the digit level, in comparison to the minimally redundant digit set equivalent. This is because the dynamic range of a larger digit set must be satisfied. As in Figure 20, greater logic complexity can be expected when constructing higher radix systems. Therefore, it is important that a SDNR/RNS configuration is selected which minimises storage requirements at the word level and logic complexity.

Radices 256 and 2048 exhibit minimal word length at maximum redundancy. Of the two radices, a digit-adder which is based on the radix 256 configuration would imply the simplest logic complexity.

Figure 23 graphically compares the word lengths of minimum and maximum digit sets for two moduli configurations.

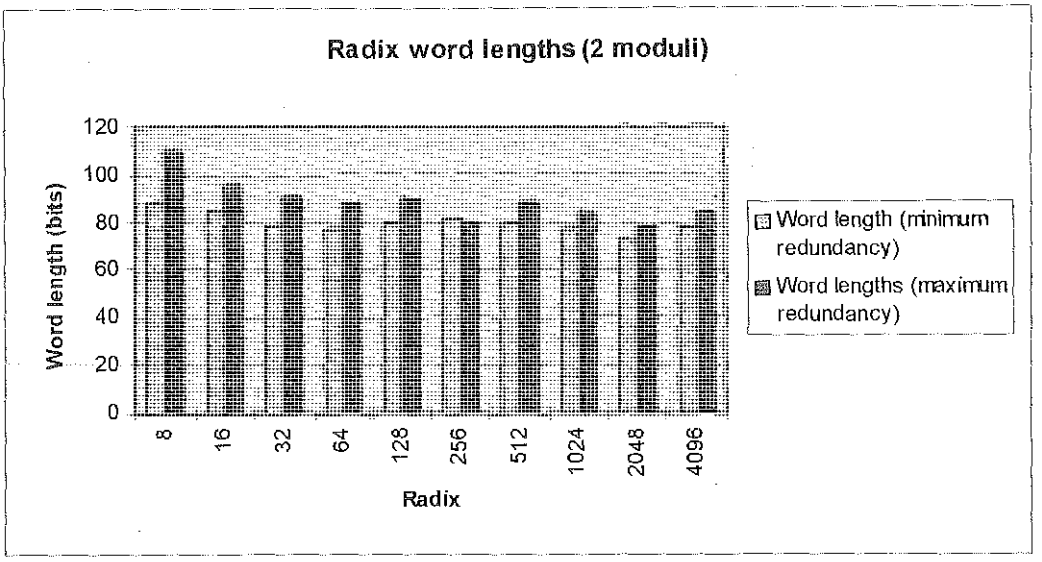

Figure 23: Radix word lengths (2 moduli).

As expected, the word lengths of the maximum redundant configurations are generally longer, compared to the corresponding minimum redundant versions. Radix 256 is an exception. For this radix, a smaller word length results when using a maximum redundant digit set. The reason for this is that for radix 256, a maximum redundant digit set achieves the required range in a lesser number of digits than the minimum redundant configuration. The required range, which is dictated by a conventional 64-bit unsigned integer, is $1.84 * 10^{19}$. For minimum redundancy, the radix 256 digit set is defined as follows:

{-129, -128, ..., -1, 0, 1, ..., 128, 129}

To satisfy the following condition:

Range (SDNR/RNS word) $\geq 2^{64} - 1$

where $2^{64} - 1 = 1.84 * 10^{19}$

the following calculation is performed to determine the number of digits required:

Number of digits	Positional weight			Positional weight range	Incremental range
	a	Radix	Exponent		
1	129	256	0	1.29E+02	1.29E+02
2	129	256	1	3.30E+04	3.32E+04
3	129	256	2	8.45E+06	8.49E+06
4	129	256	3	2.16E+09	2.17E+09
5	129	256	4	5.54E+11	5.56E+11
6	129	256	5	1.42E+14	1.42E+14
7	129	256	6	3.63E+16	3.65E+16
8	129	256	7	9.30E+18	9.33E+18
9	129	256	8	2.38E+21	2.39E+21

where Positional weight range = a * $\text{Radix}^{(\text{Exponent})}$
Incremental range = sum of positional weight ranges up to and including the current digit.

Therefore, the minimally redundant radix 256 configuration requires 9 digits. For maximum redundancy, the digit set is as follows:

{-255, -254, ..., -1, 0, 1, ..., 255}

and the number of digits required to satisfy the 64-bit range is calculated as follows:

Number of digits	Positional weight			Positional weight range	Incremental range
	a	Radix	Exponent		
1	255	256	0	2.55E+02	2.55E+02
2	255	256	1	6.53E+04	6.55E+04
3	255	256	2	1.67E+07	1.68E+07
4	255	256	3	4.28E+09	4.29E+09
5	255	256	4	1.10E+12	1.10E+12
6	255	256	5	2.80E+14	2.81E+14
7	255	256	6	7.18E+16	7.21E+16
8	255	256	7	1.84E+19	1.84E+19

where Positional weight range = a * Radix$^{(Exponent)}$
Incremental range = sum of positional weight ranges up to and including the current digit.

The maximally redundant radix 256 configuration requires only 8 digits. Note, however, that there is only a one bit difference in word length between both radix 256 configurations. This can be explained with reference to Figure 24. All maximum redundant digit sets, including the sets for radix 256, require one more storage bit at the digit level. This extra bit reduces the storage advantage that the radix 256 maximum redundant digit set has at the word level.

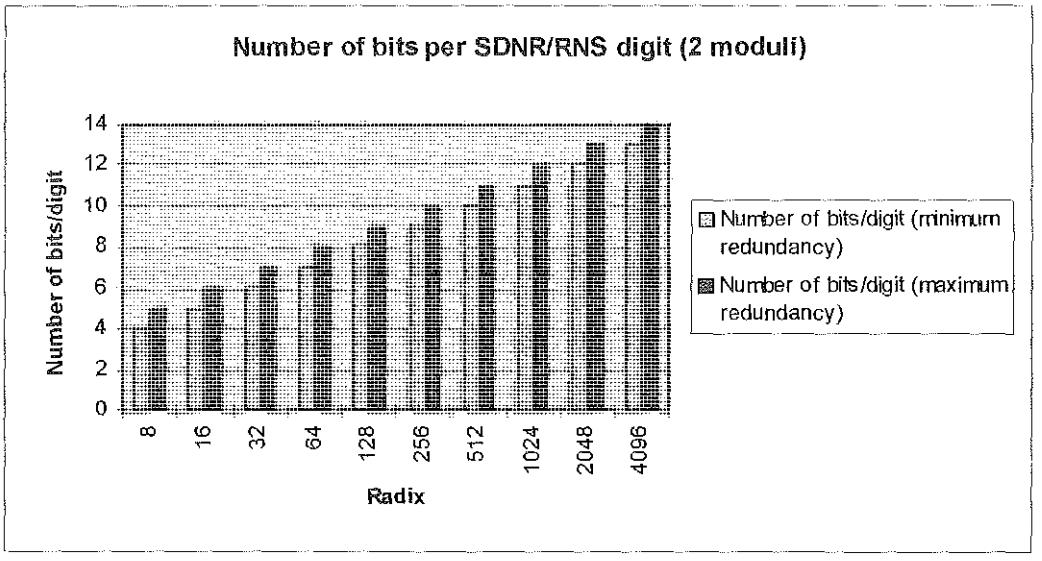

Figure 24: Number of bits per SDNR/RNS digit (2 moduli).

As previously noted, Figure 24 shows that an extra bit is required to represent SDNR/RNS systems employing 2 moduli and maximum redundancy. This is one of the main contributing factors as to why maximum redundant digit sets require comparatively more storage at the word level.

4.2.2 RNS moduli set consisting of three elements

The major reason for investigating digit-adder consisting of three modulus adders was to determine what radices would benefit from such a configuration, in terms of word lengths, and logic complexity. When analysing SDNR/RNS configurations using three moduli, it was found that the additional flexibility introduced could be used to optimise memory, or optimise adder speed. Minimum redundant digit sets will be analysed first, followed by an examination of maximum redundant sets.

4.2.2.1 Minimum redundancy

4.2.2.1.1 Minimum memory

Minimum redundant digit set configurations for radices ranging from 16 to 4096 are listed in Table 9. These configurations have been selected based on the guidelines for choosing configurations listed in a previous section, and on their minimal use of memory at the digit level and word level.

Table 9: SDNR/RNS configurations (3 moduli; minimum redundancy; minimum memory).

Radix	p1	p2	p3	Dynamic range (p1p2p3)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
16	2	3	5	30	9	17	102	6	1.84E+19	1.77E+20
32	2	5	7	70	17	13	91	7	1.84E+19	2.02E+19
64	4	5	7	140	33	11	88	8	1.84E+19	3.87E+19
128	5	7	8	280	65	10	90	9	1.84E+19	6.04E+20
256	5	7	8	280	129	9	81	9	1.84E+19	2.39E+21
512	7	8	11	616	257	8	80	10	1.84E+19	2.38E+21
1024	8	11	13	1144	513	7	77	11	1.84E+19	5.92E+20
2048	11	13	15	2145	1025	6	72	12	1.84E+19	3.69E+19
4096	15	16	19	4560	2049	6	78	13	1.84E+19	2.36E+21

4.2.2.1.2 Minimum delay

Maximum redundant digit set configurations are listed in Table 10. These configurations have been selected based on the guidelines for choosing configurations given in a previous section, and on the magnitude of the largest modulus element. By minimising the magnitude of the largest modulus element, logic complexity is minimised for that modulus adder. This, in turn, improves the speed of the digit-adder.

Table 10: SDNR/RNS configurations (3 moduli; minimum redundancy; minimum delay).

Radix	p1	p2	p3	Dynamic range (p1p2p3)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
16	2	3	5	30	9	17	102	6	1.84E+19	1.77E+20
32	2	5	7	70	17	13	91	7	1.84E+19	2.02E+19
64	4	5	7	140	33	11	88	8	1.84E+19	3.87E+19
128	5	7	8	280	65	10	90	9	1.84E+19	6.04E+20
256	5	7	8	280	129	9	81	9	1.84E+19	2.39E+21
512	7	9	10	630	257	8	88	11	1.84E+19	2.38E+21
1024	8	11	13	1144	513	7	77	11	1.84E+19	5.92E+20
2048	11	13	15	2145	1025	6	72	12	1.84E+19	3.69E+19
4096	15	16	19	4560	2049	6	78	13	1.84E+19	2.36E+21

Note that some SDNR/RNS configurations listed in Table 9 and Table 10 are identical. This is because those configurations exhibit optimal memory and delay characteristics. In fact, for minimum redundant digit sets, many of the configurations selected satisfy both optimal memory and delay requirements. Figure 25 shows word lengths for the respective radices.

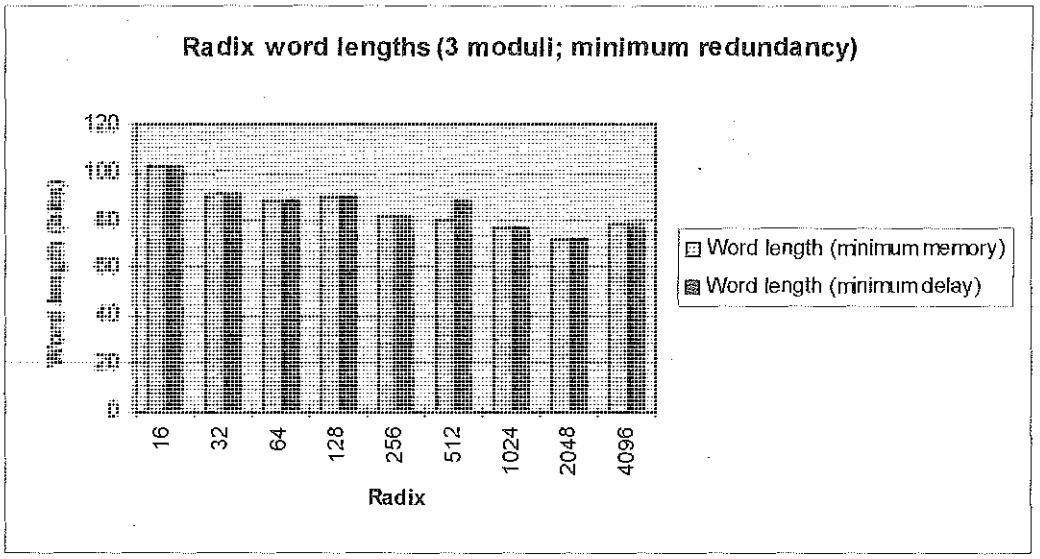

Figure 25: Radix word lengths (3 moduli; minimum redundancy).

It can be seen from Figure 25 that, in general, most of the word lengths are the same for both types of configurations. Only the configurations for radix 512 have differing word lengths. From this observation, it seems that when optimising logic delay, a tradeoff in storage requirements is necessary.

From Figure 25, observe that, like the 2 moduli cases, the trend follows a sinusoidal path with valleys and peaks corresponding to certain radices. The lower radices seem to require more storage for the word length. On the other hand, radices 1024 and 2048 indicate relatively small word lengths. From the radices compared in Figure 25, radices 1024 and 2048 seem most favourable from a storage requirements standpoint.

Figure 26 shows the number of bits required at the digit level for each radix listed in Table 9 and Table 10.

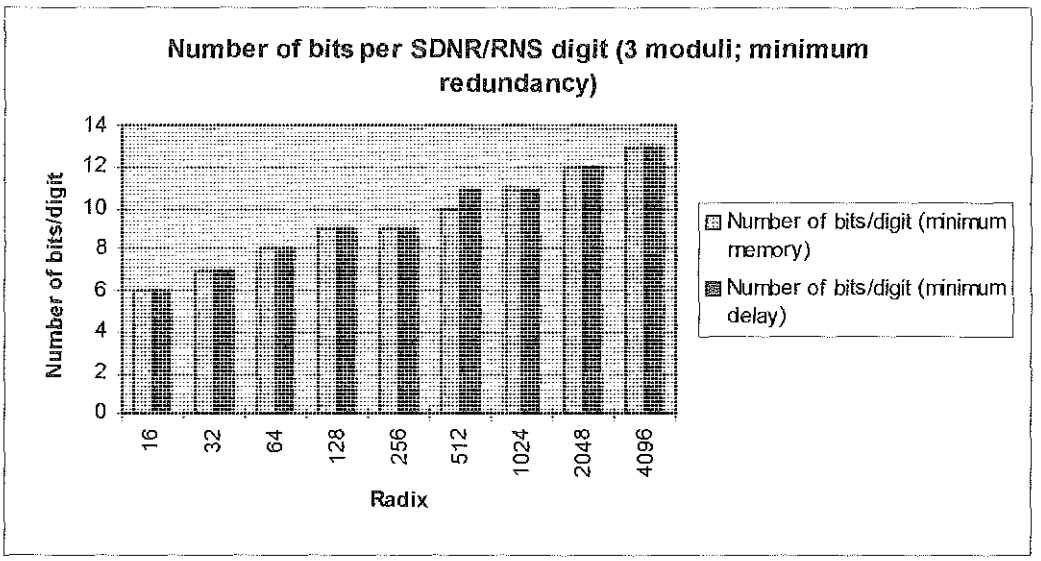

Figure 26: Number of bits per SDNR/RNS digit (3 moduli; minimum redundancy).

Only radix 512 shows any difference in the number of bits per SDNR/RNS digit required. This difference corresponds to the variance in the word length, shown in Figure 25. Notice that in Figure 26, that the trend is not log-linear over the range of radices present. The cause of this is that for radices 128 and 256, there is no increase in the number of bits per digit. This anomaly is important in that it allows the design of more efficient SDNR/RNS digit-adders. This topic will be discussed later. For now

though, it is enough to say that a correction in the number of bits per SDNR/RNS digit occurs at radix 256.

4.2.2.2 Maximum redundancy

4.2.2.2.1 Miuimum memory

Maximum redundant digit set configurations for selected radices are listed in Table 11. These configurations have been chosen based on configuration selection guidelines stated previously, and on their minimal use of memory at the digit level and word level.

Table 11: SDNR/RNS configurations (3 moduli; maximum redundancy; minimum memory).

Radix	p1	p2	p3	Dynamic range (p1p2p3)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
16	2	3	7	42	15	16	96	6	1.84E+19	1.84E+19
32	3	7	8	168	31	13	104	8	1.84E+19	3.69E+19
64	5	7	8	280	63	11	99	9	1.84E+19	7.38E+19
128	7	8	11	616	127	10	100	10	1.84E+19	1.18E+21
256	7	8	11	616	255	8	80	10	1.84E+19	1.84E+19
512	8	11	13	1144	511	8	88	11	1.84E+19	4.72E+21
1024	11	13	15	2145	1023	7	84	12	1.84E+19	1.18E+21
2048	15	16	19	4560	2047	6	78	13	1.84E+19	7.38E+19
4096	16	21	25	8400	4095	6	84	14	1.84E+19	4.72E+21

4.2.2.2.2 Minimum delay

Listed in Table 12 are maximum redundant digit set configurations for moduli ranging from 16 to 4096. These configurations have been selected based on the guidelines for choosing configurations given in a previous section, and on the minimisation of the magnitude of the largest modulus element.

Table 12: SDNR/RNS configurations (3 moduli; maximum redundancy; minimum delay).

Radix	p1	p2	p3	Dynamic range (p1p2p3)	a	Number of digits	Number of bits	Number of bits/digit	Required range	Actual range
16	3	4	5	60	15	16	112	7	1.84E+19	1.84E+19
32	3	7	8	168	31	13	104	8	1.84E+19	3.69E+19
64	5	7	8	280	63	11	99	9	1.84E+19	7.38E+19
128	7	9	10	630	127	10	110	11	1.84E+19	1.18E+21
256	7	9	10	630	255	8	88	11	1.84E+19	1.84E+19
512	8	11	13	1144	511	8	88	11	1.84E+19	4.72E+21
1024	11	13	15	2145	1023	7	84	12	1.84E+19	1.18E+21
2048	15	16	19	4560	2047	6	78	13	1.84E+19	7.38E+19
4096	17	21	23	8211	4095	6	90	15	1.84E+19	4.72E+21

Some of the configurations listed in Table 11 and Table 12 are the same for the corresponding radix. As in the case for minimum redundancy, the reason for this is that those configurations are optimised in terms of both memory and speed. The word lengths for those configurations listed in the tables for maximum redundancy are shown in Figure 27.

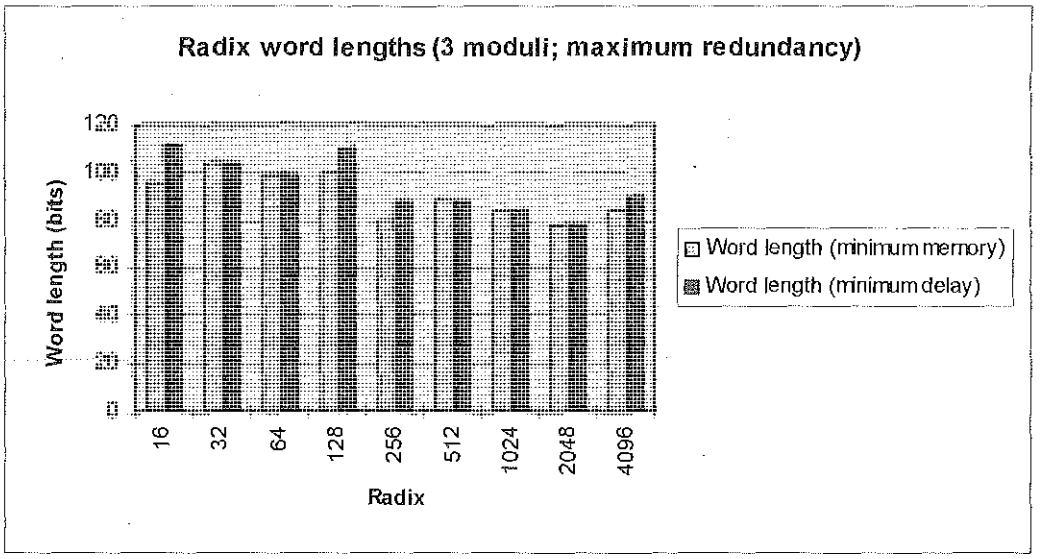

Figure 27: Radix word lengths (3 moduli; maximum redundancy).

Radices 16, 128, 256, and 4096 have a unique configuration for optimising memory and delay. The other radices have a configuration that optimises both memory and delay at the same time. Notice that for the radices with two unique configurations, minimising delay requires the greater amount of storage capacity. Also, the radices 16 to 128 seem to be less efficient in terms of word length representation compared to the radices 256 through to 4096. Figure 28 shows storage requirements at the digit level for each radix.

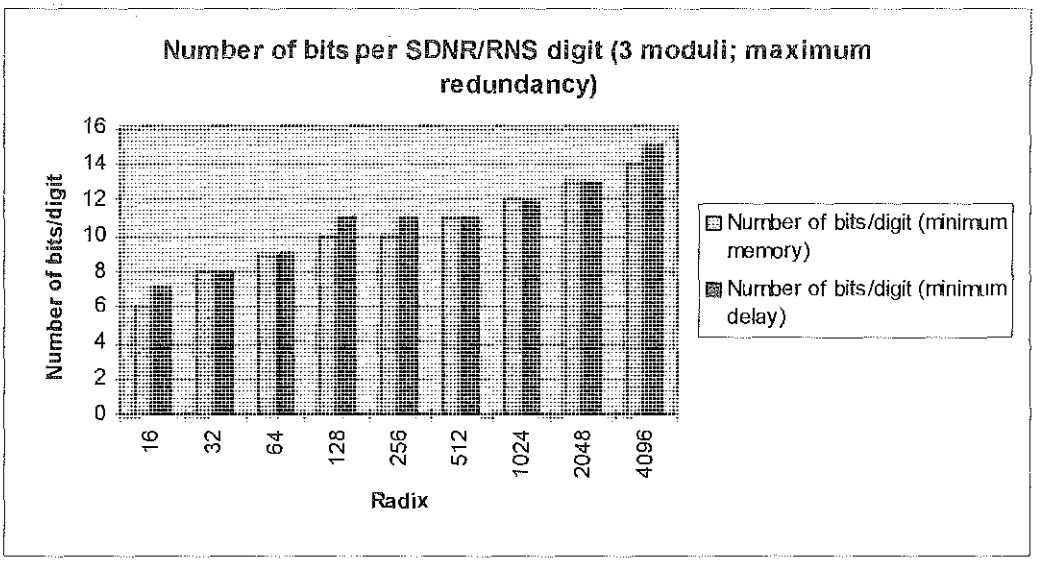

Figure 28: Number of bits per SDNR/RNS digit (3 modnli; maximum redundancy).

As in Figure 26, the graph shown in Figure 28 does not show a log-linear trend for either set of data. For the configurations which focus on minimising memory requirements, a correction takes place at radix 256. For the configurations which optimise delay, the number of bits per digit stays constant over radices 128, 256, and 512. Therefore, a digit size correction occurs at radix 512. These corrections give a clue as to when a three moduli digit-adder would give a more efficient implementation, when compared to a two moduli digit-adder.

Figure 29, Figure 30, Figure 31, and Figure 32 have been included for completeness. These graphs show that, for three moduli SDNR/RNS configurations, minimum redundant digit sets use memory more efficiently than maximum redundant sets.

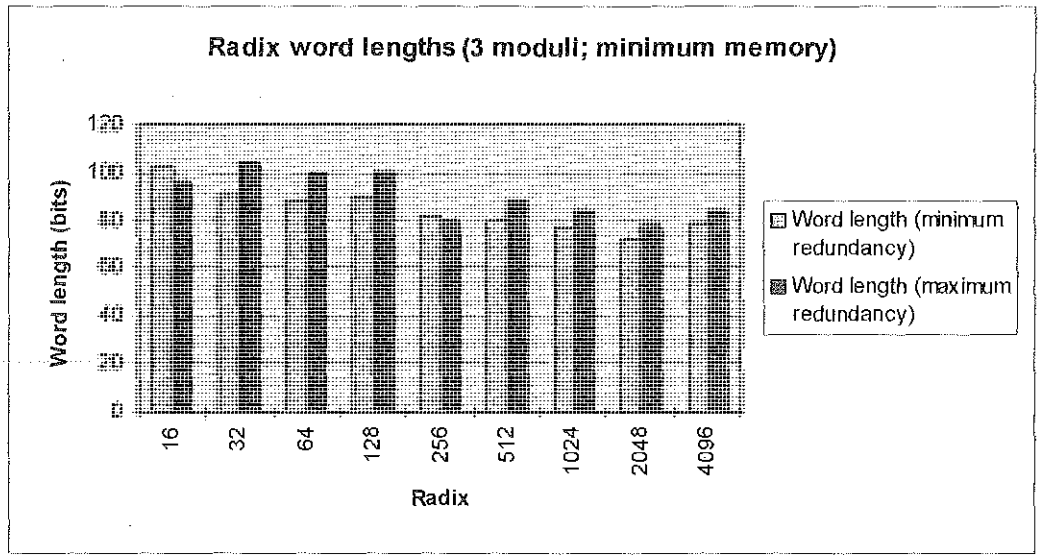

Figure 29: Radix word lengths (3 moduli; minimum memory).

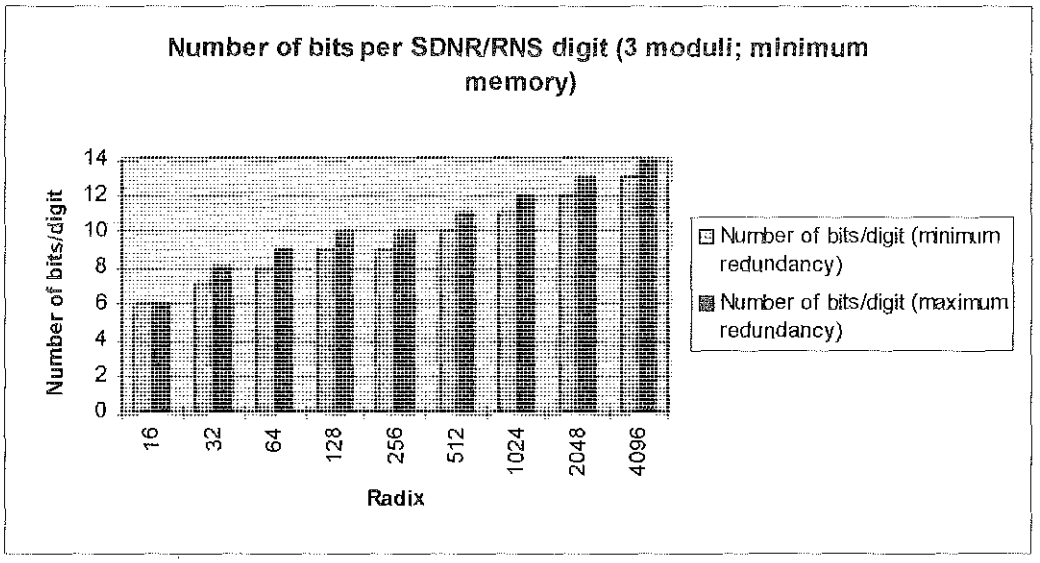

Figure 30: Number of bits per SDNR/RNS digit (3 moduli; minimum memory).

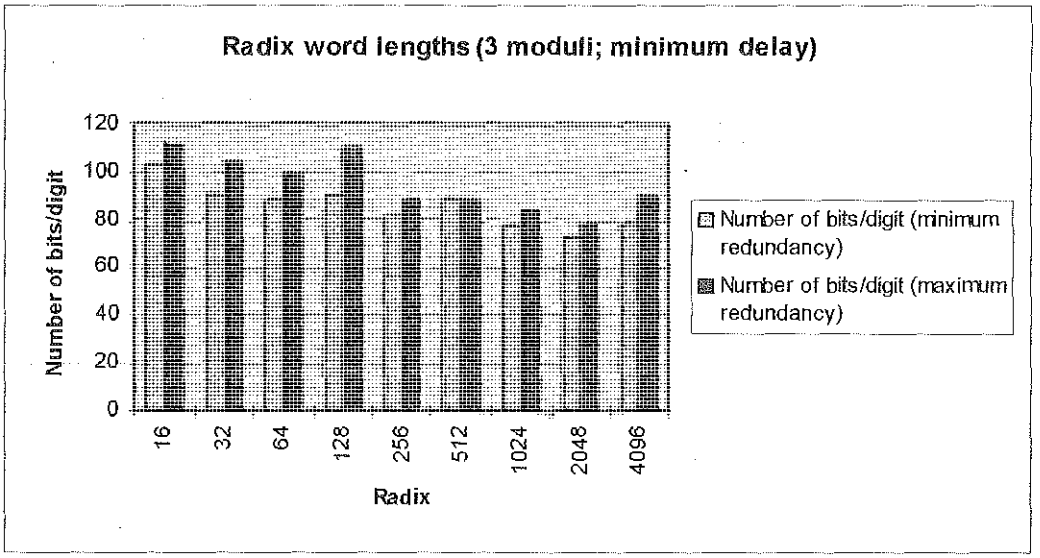

Figure 31: Radix word lengths (3 moduli; minimum delay).

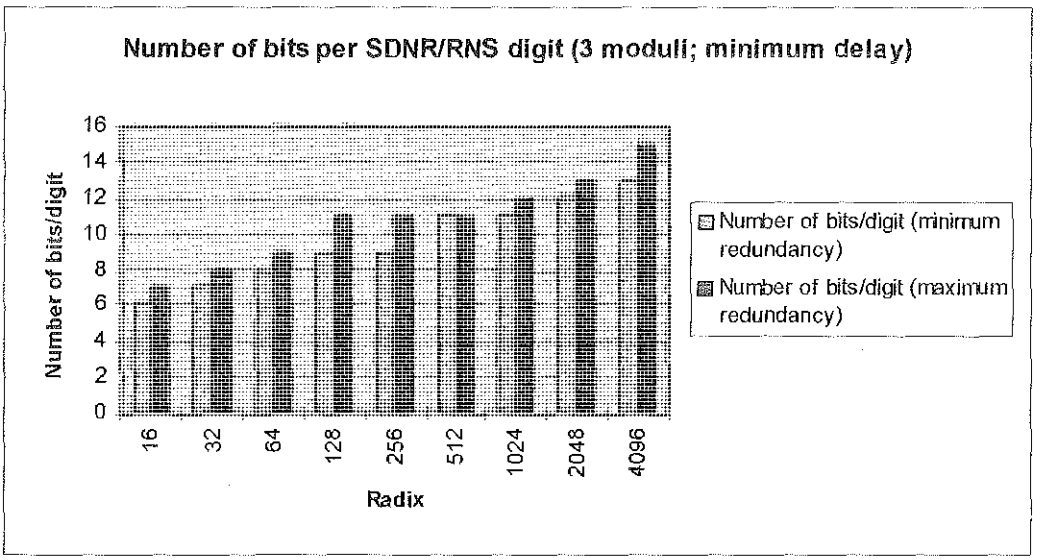

Figure 32: Number of bits per SDNR/RNS digit (3 moduli; minimum delay).

4.2.3 Comparisons

This section compares the relative merits of two moduli digit-adders and three-moduli digit-adders. Following this analysis is an additional guideline regarding what kind of digit-adder should be used for greatest efficiency. Figure 33 compares word lengths for various SDNR/RNS configurations.

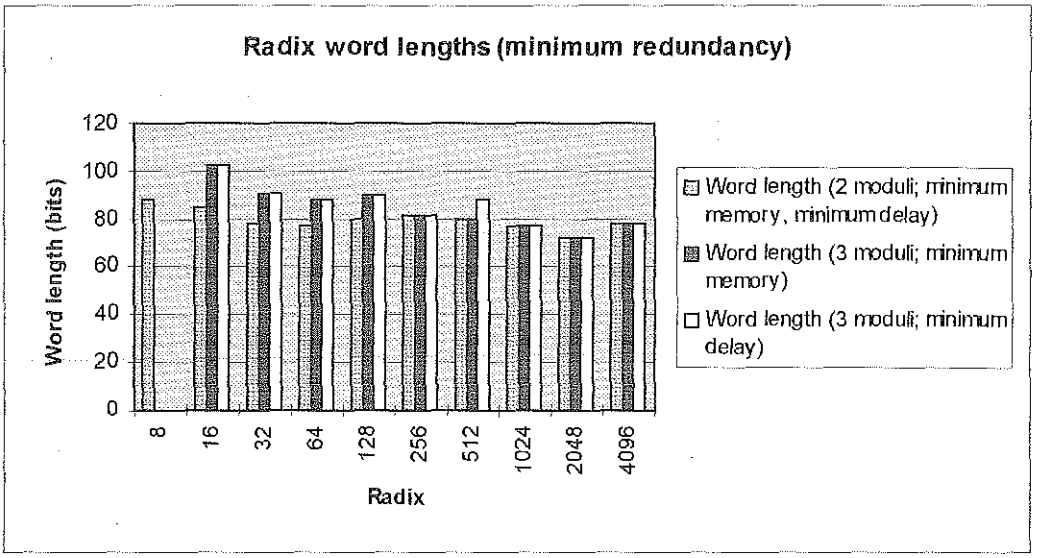

Figure 33: Radix word lengths (minimum redundancy).

Recall that for three moduli configurations, a correction in the number of bits per digit took place at radix 256. From Figure 33, it is clear that this correction coincides with the fact that storage critical three moduli systems match two moduli configurations, in terms of memory requirements, for radices greater than or equal to 256, and less than or equal to 4096. The following example illustrates that for the higher radices, a three moduli set digit-adder is more efficient in operation. The two moduli configuration for radix 256 includes the moduli set (16, 17). In comparison, the corresponding three moduli system includes the moduli set (5, 7, 8). The largest modulus value for the two moduli configuration is 17. This means that the modulus adder, in this case, must accept 17 values. On the other hand, the most complicated modulus adder in the three moduli system must accept only 8 values. This is a significant difference in the number of values both modulus adders must compute. The modulus 17 adder would be inherently more complex and slower than the modulus 8 adder. Therefore, at the expense of an additional modulus adder, the three moduli digit-adder would be able to add or subtract faster than if only a two moduli configuration was used. Similar cases arise for radices 512, 1024, 2048, and 4096.

Notice that in Figure 33, the radix 2048 minimum redundant configuration results in the smallest word length of 72 bits. However, to achieve such a small word length, 12 bits per digit are required (Figure 34). In comparison, the two moduli, radix 32, minimum redundant configuration requires 78 bits, and only 6 bits per digit. Therefore, by using the radix 32 configuration, the complexity at the digit level can be halved by increasing the word length by 6 bits, or 8.33%. Similarly, the two moduli, radix 64, minimum redundant configuration requires 77 bits at the word level at 7 bits per digit. This radix 64 configuration, in comparison to the cited radix 32 configuration, reduces the storage requirements at the word level by 1 bit (a decrease of 1.28%), but requires 1 more bit at the digit level (an increase of 16.67%). The favourable characteristics of the radix 32 configuration seems to make it a good choice for an attempt at constructing a VLSI digit-adder. The minimum redundancy will ensure a relatively simple adder implementation.

Shown in Figure 34 is a graph comparing the number of bits per SDNR/RNS digit for each radix.

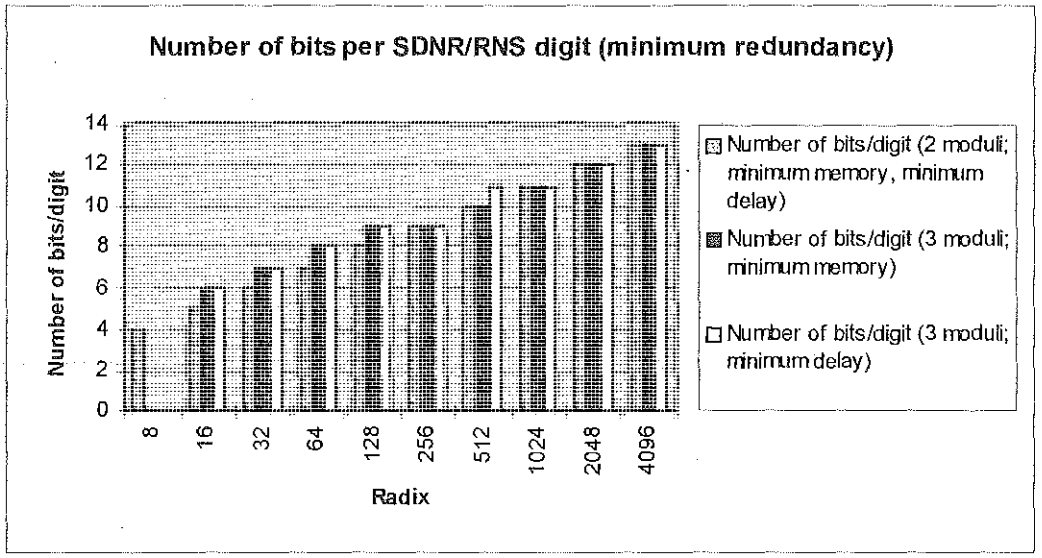

Figure 34: Nnmber of bits per SDNR/RNS digit (minimum redundancy).

Figure 34 shows that the correction at radices 256 and 512 for the three moduli configurations keeps memory requirements at a par with the corresponding two moduli representations at the higher radices.

For completeness, graphical comparisons for maximum redundant two and three moduli configurations are shown in Figure 35 and Figure 36. Note that similar conclusions can be drawn about maximum redundant representations as for minimum redundant representations, in terms of whether to use a digit-adder based on two or three moduli sets.

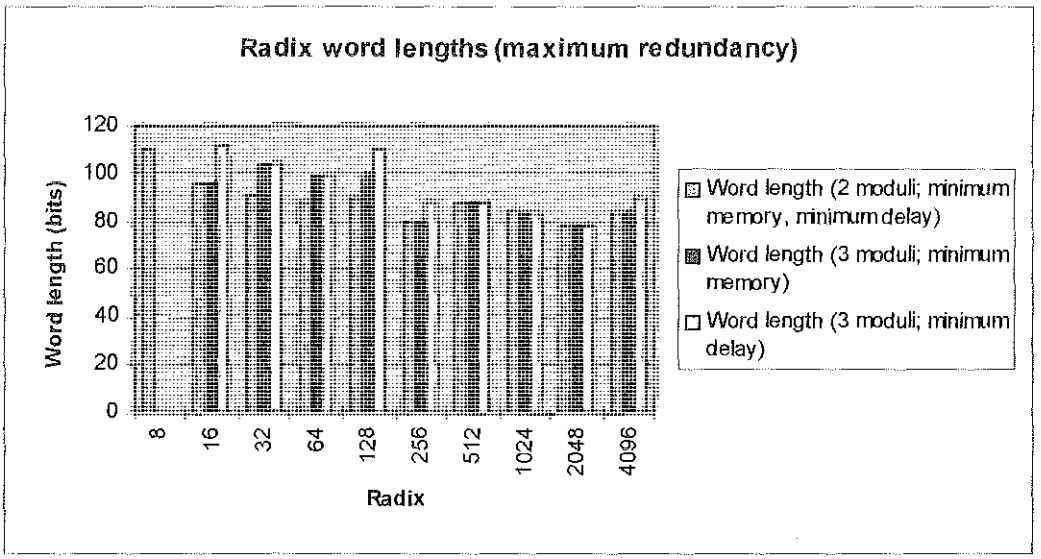

Figure 35: Radix word lengths (maximum redundancy).

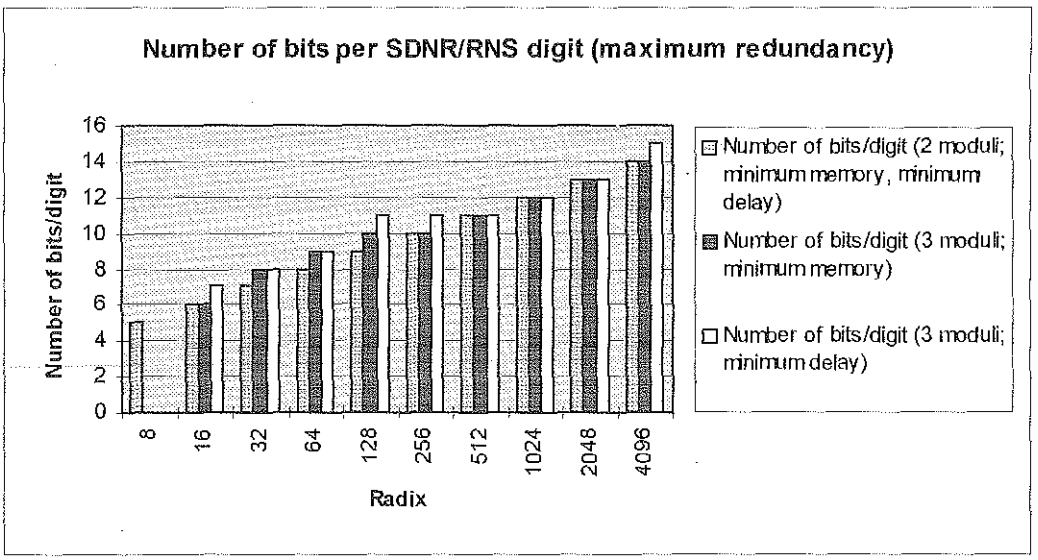

Figure 36: Number of bits per SDNR/RNS digit (maximum redundancy).

To represent a conventional 64-bit unsigned integer, a two moduli configuration should be chosen for a digit-adder if the radix is less than 256. Otherwise, if a radix greater than or equal to 256 is selected, then a three moduli configuration would be preferable.

For the digit adder implemented, the minimum redundant radix 32 configuration listed in Table 7 was used. This configuration exhibits efficient memory characteristics, as well as maintaining relatively low logic complexity, when used to represent a conventional 64-bit unsigned integer. The choice of configuration can be described in terms of the guidelines discussed in section 4.1-SDNR/RNS configuration analysis:

For the radix guidelines:

- The chosen configuration has a word length of 78 bits at 6 bits per digit. This resulted in a relatively simple digit-adder.
- Radix 32 is a power of 2.

For the digit set guidelines:

- A minimum redundant digit set was chosen to keep storage requirements and logic complexity small.
- A nondisjoint digit set was selected so that the greatest dynamic range at the digit level could be achieved.

For the moduli guidelines:

- The moduli 5 and 7 are relatively prime.
- The moduli 5 and 7 are as small as possible so that operations required minimum computational time.
- The product of the moduli was large enough in order to implement the desired dynamic range.
- The moduli 5 and 7 result in a balanced decomposition of the dynamic range. Three bits are required to represent each moduli. There is no difference between the number of bits to represent the different moduli.

By using the set theory of arithmetic decomposition, the addition process using the chosen configuration can be verified as follows:

Stage 1: $32\langle 2^1\rangle + \langle 32^{16}\rangle \Leftarrow \langle 34^{17}\rangle + \langle 34^{17}\rangle$

Stage 2: $\langle 34^{17}\rangle \Leftarrow \langle 32^{16}\rangle + \langle 2^1\rangle$

5. Design

This chapter includes the logic design for the SDNR/RNS digit adder, the conversion circuits, and the sign detector circuit. Each of these arithmetic components involved defining truth tables, karnaugh maps, and then deriving logic equations. A software simulation was written, so that each arithmetic component could be tested at the unit and system levels.

For this project, the logic minimisation method used was based on traditional karnaugh map reduction. Karnaugh maps were used because the number of inputs into a logic gate never exceeded 6. It was found that 6 variable karnaugh maps were about the upper bound limit in determining the minimal sums of products expression for a particular output.

If a circuit, for example, a digit adder, which contained logic gates with more than 6 inputs was required, then other logic minimisation methods would have been used. Hayes (1993) states that the visual identification and selection of prime terms via karnaugh maps becomes more difficult as the number of inputs in a function increase. Hayes (1993) goes on to detail alternative logic minimisation schemes. He discusses the tabular, or Quine-McCluskey method of logic minimisation, which is suitable for solving large input problems with the aid of a computer. Kuczborski (1993) makes use of the Quine-McCluskey minimisation method. Hayes (1993) also covers approximate, or heuristic minimisation methods, and the problem of designing minimum-cost multilevel circuits. Furthermore, with such large numbers of inputs into each logic gate, logic decomposition would be recommended to reduce the number of inputs into each gate to between 2 and 5.

5.1 SDNR/RNS configuration

Table 13 defines the configuration (from the last chapter) which was used for the design of a SDNR/RNS arithmetic system.

Table 13: SDNR/RNS configuration.

Radix	p1	p2	Dynamic range	a	Number SDNR/RNS digits to represent 64-bit integer	Number SDNR/RNS bits to represent 64-bit integer
32	5	7	35	17	13	78

The system was based on radix 32. The main reason for such a choice of radix was that a conventional binary number can be easily converted to the SDNR/RNS notation by grouping bits. In this case, a 5 (= $\log_2 32$) bit segment in a conventional binary number can be directly converted to a SDNR/RNS digit. For this particular configuration, 78 bits are required to represent a 64-bit conventional number.

A minimally redundant digit set (-17, 16, ..., -1, 0, +1, ..., 16, +17) was selected based on the choice of moduli pair. A minimally redundant digit set was used because the particular combination of the chosen moduli pair and digit set allowed for optimal nondisjoint sets. Nondisjoint sets, in comparison to disjoint sets using the same set of moduli, increase the dynamic range of SDNR/RNS digits. Table 14 lists the nondisjoint digit set used for the arithmetic system.

Table 14: Nondisjoint digit set.

Digit	SDNR/RNS digit			Digit	SDNR/RNS digit		
	mod p1	mod p2			mod p1	mod p2	
							Region 1
0	0	0					
1	1	1	carry =	-34	1	1	carry =
2	2	2	0	-33	2	2	-1
3	3	3		-32	3	3	
4	4	4		-31	4	4	
5	0	5		-30	0	5	
6	1	6		-29	1	6	
7	2	0		-28	2	0	
8	3	1		-27	3	1	
9	4	2		-26	4	2	
10	0	3		-25	0	3	
11	1	4		-24	1	4	
12	2	5		-23	2	5	
13	3	6		-22	3	6	
14	4	0		-21	4	0	
15	0	1		-20	0	1	
16	1	2		-19	1	2	
17	2	3	carry =	-18	2	3	Region 2
18	3	4	1	-17	3	4	
19	4	5		-16	4	5	carry = Region 3
20	0	6		-15	0	6	0
21	1	0		-14	1	0	
22	2	1		-13	2	1	
23	3	2		-12	3	2	
24	4	3		-11	4	3	
25	0	4		-10	0	4	
26	1	5		-9	1	5	
27	2	6		-8	2	6	
28	3	0		-7	3	0	
29	4	1		-6	4	1	
30	0	2		-5	0	2	
31	1	3		-4	1	3	
32	2	4		-3	2	4	
33	3	5		-2	3	5	
34	4	6		-1	4	6	

5.2 Reference tables

Table 15 lists equivalent decimal and binary values used throughout the design chapter.

Table 15: Decimal/binary reference table.

Decimal reference		
Digit	Digit MOD p1	Digit MOD p2
-34	1	1
-33	2	2
-32	3	3
-31	4	4
-30	0	5
-29	1	6
-28	2	0
-27	3	1
-26	4	2
-25	0	3
-24	1	4
-23	2	5
-22	3	6
-21	4	0
-20	0	1
-19	1	2
-18	2	3
-17	3	4
-16	4	5
-15	0	6
-14	1	0
-13	2	1
-12	3	2
-11	4	3
-10	0	4
-9	1	5
-8	2	6
-7	3	0
-6	4	1
-5	0	2
-4	1	3
-3	2	4
-2	3	5
-1	4	6
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	0	5
6	1	6
7	2	0
8	3	1
9	4	2
10	0	3
11	1	4
12	2	5
13	3	6
14	4	0
15	0	1
16	1	2
17	2	3
18	3	4
19	4	5
20	0	6

Binary reference		
Digit	Digit MOD p1	Digit MOD p2
11011110	001	001
11011111	010	010
11100000	011	011
11100001	100	100
11100010	000	101
11100011	001	110
11100100	010	000
11100101	011	001
11100110	100	010
11100111	000	011
11101000	001	100
11101001	010	101
11101010	011	110
11101011	100	000
11101100	000	001
11101101	001	010
11101110	010	011
11101111	011	100
11110000	100	101
11110001	000	110
11110010	001	000
11110011	010	001
11110100	011	010
11110101	100	011
11110110	000	100
11110111	001	101
11111000	010	110
11111001	011	000
11111010	100	001
11111011	000	010
11111100	001	011
11111101	010	100
11111110	011	101
11111111	100	110
00000000	000	000
00000001	001	001
00000010	010	010
00000011	011	011
00000100	100	100
00000101	000	101
00000110	001	110
00000111	010	000
00001000	011	001
00001001	100	010
00001010	000	011
00001011	001	100
00001100	010	101
00001101	011	110
00001110	100	000
00001111	000	001
00010000	001	010
00010001	010	011
00010010	011	100
00010011	100	101
00010100	000	110

Table 13 (continued): Decimal/binary reference table.

Decimal reference		
Digit	Digit MOD p1	Digit MOD p2
21	1	0
22	2	1
23	3	2
24	4	3
25	0	4
26	1	5
27	2	6
28	3	0
29	4	1
30	0	2
31	1	3
32	2	4
33	3	5
34	4	6

Binary reference		
Digit	Digit MOD p1	Digit MOD p2
00010101	001	000
00010110	010	001
00010111	011	010
00011000	100	011
00011001	000	100
00011010	001	101
00011011	010	110
00011100	011	000
00011101	100	001
00011110	000	010
00011111	001	011
00100000	010	100
00100001	011	101
00100010	100	110

Truth table variables are designated alphabetical characters as shown in Table 16 to Table 18.

Table 16: 4 variable truth table.

Input variables	Output
d c b a	⋮

Table 17: 5 variable truth table.

Input variables	Output
e d c b a	⋮

Table 18: 6 variable truth table.

Input variables	Output
f e d c b a	⋮

The truth table variables are projected into a corresponding Karnaugh map as shown in Table 19 to Table 21. Negated variables are prefixed with a '~' symbol. Furthermore, karnaugh map entries marked with an 'x' indicate a "don't care" condition.

Table 19: 4 variable Karnaugh map.

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

Table 20: 5 variable Karnaugh map.

~e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

Table 21: 6 variable Karnaugh map.

~f ~e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

~f e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

f e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

f ~e

	~b ~a	~b a	b a	b ~a
~d ~c				
~d c				
d c				
d ~c				

5.3 Adder design

The digit adder design needed to be able to handle nondisjoint sets. Kuczborski (1993) deals with this issue, and it is his adder design that was adopted. Figure 37 shows the schematic of Kuczborski's SDNR/RNS digit adder.

A favourable characteristic of Kuczborski's adder is modularity. To add two SDNR/RNS numbers, both representing 64-bit conventional numbers, 13 digit adders are required to be connected and placed side by side.

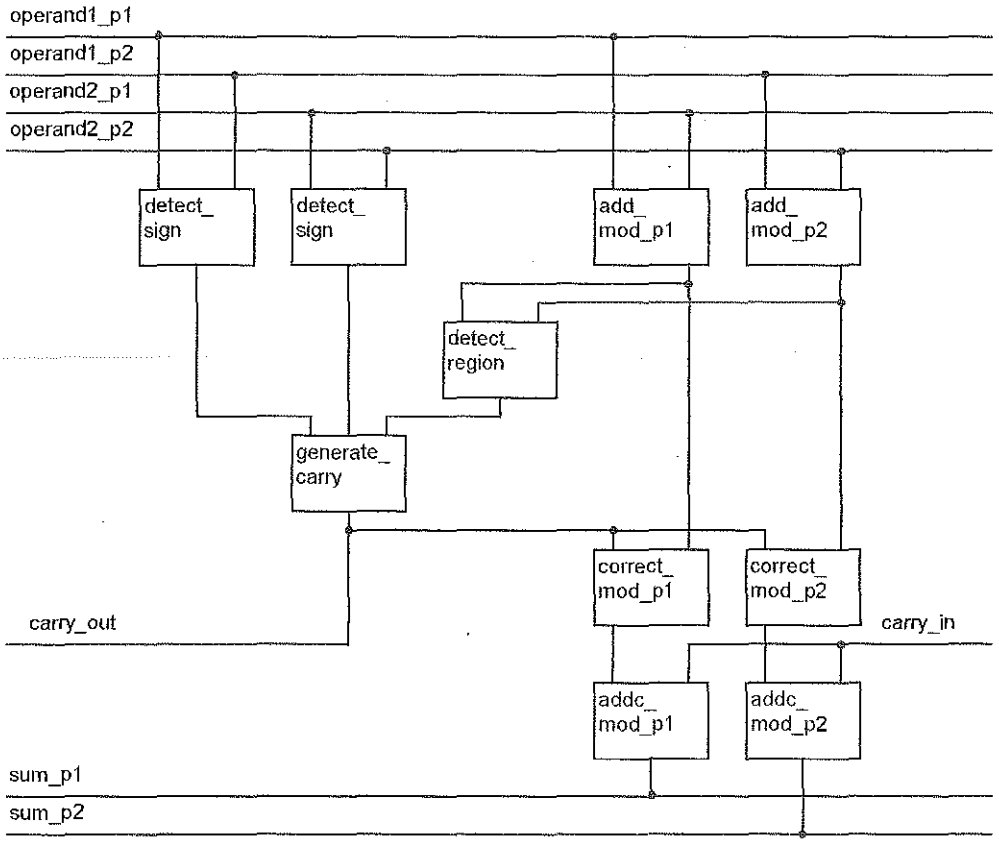

Figure 37: SDNR/RNS digit adder.

5.4 Component design

5.4.1 detect_sign

5.4.1.1 Purpose

The detect_sign component determines the sign of the respective SDNR/RNS digit.

5.4.1.2 Inputs

operand_p1: Modulus 5 number (3 bits).
operand_p2: Modulus 7 number (3 bits).

5.4.1.3 Outputs

sign_operand: Sign of the SDNR/RNS digit defined by <p1, p2> (1 bit).

5.4.1.4 Notes

The following algorithm is used to determine the output of the detect_sign component:

IF (SDNR/RNS digit <p1, p2> is positive) THEN
    sign_operand = 0

ELSE

sign_operand = 1

ENDIF

For operand_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations. For operand_p2, the binary combination 111 does not exist. Modulus 7 does not allow this combination.

5.4.1.5 Truth table

operand_p1	operand_p2	sign_ operand
011	100	1
100	101	1
000	110	1
001	000	1
010	001	1
011	010	1
100	011	1
000	100	1
001	101	1
010	110	1
011	000	1
100	001	1
000	010	1
001	011	1
010	100	1
011	101	1
100	110	1
000	000	0
001	001	0
010	010	0
011	011	0
100	100	0
000	101	0
001	110	0
010	000	0
011	001	0
100	010	0
000	011	0
001	100	0
010	101	0
011	110	0
100	000	0
000	001	0
001	010	0
010	011	0

5.4.1.6 Karnaugh maps

00

	00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	0	1	x	0
10	1	0	1	0

01

	00	01	11	10
00	0	1	0	0
01	1	0	x	1
11	1	1	x	0
10	1	0	0	1

11

	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

	00	01	11	10
00	0	1	1	0
01	0	1	x	1
11	x	x	x	x
10	x	x	x	x

5.4.1.7 Logic equations

sign =

a & f
+ b & c & f
+ ~a & c & ~d & ~f
+ a & c & d & ~f
+ ~a & ~b & c & e
+ ~a & ~c & d & e
+ ~a & b & ~d & ~e & ~f
+ a & b & d & ~e & ~f
+ a & ~b & ~c & ~d & e
+ ~a & ~b & ~c & d & ~e & ~f

5.4.2 detect_region

5.4.2.1 Purpose

The detect_region component determines the locality of the intermediate sum within the disjoint sets. The disjoint sets are broken up into 3 distinct localities, or regions. Refer to Table 14 for a diagrammatic representation of the regions.

5.4.2.2 Inputs

uncorrected_sum_p1: Modulus 5 intermediate sum (3 bits).
uncorrected_sum_p2: Modulus 7 intermediate sum (3 bits).

5.4.2.3 Outputs

region: Intermediate sum region (2 bits).

5.4.2.4 Notes

The following algorithm is used to determine the output of the detect_region component (the numbers 1, 2, and 3 are depicted in Table 14):

```
CASEOF region
        1:
                region = 00
        2:
                region = 01
        3:
                region = 10
ENDCASE
```

For uncorrected_sum_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations. For uncorrected_sum_p2, the binary combination 111 does not exist. Modulus 7 does not allow this combination.

5.4.2.5 Truth table

uncorrected sum p1	uncorrected sum p2	region (b1 b0)
000	000	00
000	001	00
000	010	10
000	011	00
000	100	10
000	101	00
000	110	10
001	000	10
001	001	00
001	010	00
001	011	10
001	100	00
001	101	10
001	110	00
010	000	00
010	001	10
010	010	00
010	011	01
010	100	10
010	101	00
010	110	10
011	000	10
011	001	00
011	010	10
011	011	00
011	100	01
011	101	10
011	110	00
100	000	00
100	001	10
100	010	00
100	011	10
100	100	00
100	101	10
100	110	10

5.4.2.6 Karnaugh maps

00

b1 \	00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	0	1	x	0
10	1	0	1	0

01

b1 \	00	01	11	10
00	0	1	0	0
01	1	0	x	1
11	0	1	x	0
10	1	0	0	1

11

b1 \	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b1 \	00	01	11	10
00	0	1	1	0
01	0	1	x	1
11	x	x	x	x
10	x	x	x	x

00

b0 \	00	01	11	10
00	0	0	0	0
01	0	0	x	0
11	0	0	x	0
10	0	0	0	0

01

b0 \	00	01	11	10
00	0	0	1	0
01	0	0	x	0
11	1	0	x	0
10	0	0	0	0

11

b0 \	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b0 \	00	01	11	10
00	0	0	0	0
01	0	0	x	0
11	x	x	x	x
10	x	x	x	x

5.4.2.7 Logic equations

```
region_b1 =
        a & f
        + a & c & d
        + b & c & f
        + a & b & d & ~e
        + ~a & c & ~d & ~f
        + ~a & ~b & ~c & d
        + ~a & ~c & d & e
        + ~a & b & ~d & ~e & ~f
        + a & ~b & ~c & ~d & e

region_b0 =
        a & b & ~d & e
        + ~a & ~b & c & d & e
```

5.4.3 generate_carry

5.4.3.1 Purpose

The generate_carry component determines if a carry is required. The output of this component is based upon the signs of both SDNR/RNS digits, and the region of the intermediate sum.

5.4.3.2 Inputs

sign_operand1: Sign of the first SDNR/RNS digit operand (1 bit).
sign_operand2: Sign of the second SDNR/RNS digit operand (1 bit).
region: Intermediate sum region (2 bits).

5.4.3.3 Outputs

carry_out: Carry value for correction of intermediate sum (2 bits).

5.4.3.4 Notes

The following algorithm is used to determine the output of the generate_carry component (the numbers -1, 0, and 1 are depicted in Table 14):

```
CASEOF carry_out
        -1:
                carry_out = 00
        0:
                carry_out = 01
        1:
                carry_out = 10
ENDCASE
```

For region, the binary combination 11 does not exist.

5.4.3.5 Truth table

sign_operand1	sign_operand2	region	carry_out (b1 b0)
0	0	00	01
0	0	01	10
0	0	10	10
0	1	00	01
0	1	01	00
0	1	10	01
1	0	00	01
1	0	01	00
1	0	10	01
1	1	00	00
1	1	01	00
1	1	10	01

5.4.3.6 Karnaugh maps

b1	00	01	11	10
00	0	1	x	1
01	0	0	x	0
11	0	0	x	0
10	0	0	x	0

b0	00	01	11	10
00	1	0	x	0
01	1	0	x	1
11	0	0	x	1
10	1	0	x	1

5.4.3.7 Logic equations

carry_out_b1 =
a & ~c & ~d
+ b & ~c & ~d

carry_out_b0 =
b & c
+ ~a & ~b & ~d
+ ~a & ~c & d

5.4.4 add_mod_p1

5.4.4.1 Purpose

The add_mod_p1 component adds the p1 (modulus 5) moduli of the SDNR/RNS input operands. add_mod_p1 outputs the p1 modulus of the intermediate sum.

5.4.4.2 Inputs

operand1_p1: Modulus 5 segment of the first SDNR/RNS digit operand (3 bits).
operand2_p1: Modulus 5 segment of the second SDNR/RNS digit operand (3 bits).

5.4.4.3 Outputs

uncorrected_sum_p1: Modulus 5 segment of the intermediate sum (3 bits).

5.4.4.4 Notes

For operand1_p1 and operand2_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations.

5.4.4.5 Truth table

operand1_p1	operand2_p1	uncorrected_sum_p1 (b2 b1 b0)
000	000	000
000	001	001
000	010	010
000	011	011
000	100	100
001	000	001
001	001	010
001	010	011
001	011	100
001	100	000
010	000	010
010	001	011
010	010	100
010	011	000
010	100	001
011	000	011
011	001	100
011	010	000
011	011	001
011	100	010
100	000	100
100	001	000
100	010	001
100	011	010
100	100	011

5.4.4.6 Karnaugh maps

00

b2	00	01	11	10
00	0	0	0	0
01	1	x	x	x
11	0	x	x	x
10	0	0	1	0

01

b2	00	01	11	10
00	0	0	0	1
01	0	x	x	x
11	0	x	x	x
10	0	1	0	0

11

b2	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b2	00	01	11	10
00	1	0	0	0
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

00

b1	00	01	11	10
00	0	0	1	1
01	0	x	x	x
11	0	x	x	x
10	0	1	0	1

01

b1	00	01	11	10
00	1	1	0	0
01	0	x	x	x
11	1	x	x	x
10	1	0	0	0

11

b1	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b1	00	01	11	10
00	0	0	1	0
01	1	x	x	x
11	x	x	x	x
10	x	x	x	x

00

b0 \ 00	01	11	10
00	0	1	1	0
01	0	x	x	x
11	0	x	x	x
10	1	0	0	1

01

b0 \ 00	01	11	10
00	0	1	0	0
01	1	x	x	x
11	0	x	x	x
10	1	0	1	0

11

b0 \ 00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b0 \ 00	01	11	10
00	0	0	0	1
01	1	x	x	x
11	x	x	x	x
10	x	x	x	x

5.4.4.7 Logic equations

uncorrected_sum_p1_b2 =
~a & ~b & ~c & f
+ ~a & b & ~d & e
+ a & ~b & d & e
+ a & b & d & ~e
+ c & ~d & ~e & ~f

uncorrected_sum_p1_b1 =
c & f
+ a & b & f
+ a & ~b & d & ~e
+ b & ~d & ~e & ~f
+ ~a & b & ~e & ~f
+ ~b & ~c & ~d & e
+ ~a & ~b & d & e

uncorrected_sum_p1_b0 =
c & f
+ c & ~d & e
+ ~a & b & f
+ ~a & ~c & d & ~e
+ a & ~d & ~e & ~f
+ a & ~b & ~d & e
+ a & b & d & e
+ ~a & ~b & ~c & d & e

5.4.5 add_mod_p2

5.4.5.1 Purpose

The add_mod_p2 component adds the p2 (modulus 7) moduli of the SDNR/RNS input operands. add_mod_p2 outputs the p2 modulus of the intermediate sum.

5.4.5.2 Inputs

operand1_p2: Modulus 7 segment of the first SDNR/RNS digit operand (3 bits).
operand2_p2: Modulus 7 segment of the second SDNR/RNS digit operand (3 bits).

5.4.5.3 Outputs

uncorrected_sum_p2: Modulus 7 segment of the intermediate sum (3 bits).

5.4.5.4 Notes

For operand1_p2 and operand2_p2, the binary combination 111 does not exist. Modulus 7 does not allow this combination.

5.4.5.5 Truth table

operand1_p2	operand2_p2	uncorrected_sum_p2 (b2 b1 b0)
000	000	000
000	001	001
000	010	010
000	011	011
000	100	100
000	101	101
000	110	110
001	000	001
001	001	010
001	010	011
001	011	100
001	100	101
001	101	110
001	110	000
010	000	010
010	001	011
010	010	100
010	011	101
010	100	110
010	101	000
010	110	001
011	000	011
011	001	100
011	010	101
011	011	110
011	100	000
011	101	001
011	110	010
100	000	100
100	001	101
100	010	110
100	011	000
100	100	001
100	101	010
100	110	011
101	000	101
101	001	110
101	010	000
101	011	001
101	100	010
101	101	011
101	110	100
110	000	110
110	001	000
110	010	001
110	011	010
110	100	011
110	101	100
110	110	101

5.4.5.6 Karnaugh maps

00

b2	00	01	11	10
00	0	0	0	0
01	1	1	x	1
11	1	1	x	0
10	0	0	1	0

01

b2	00	01	11	10
00	0	0	1	1
01	1	0	x	0
11	0	0	x	0
10	0	1	1	1

11

b2	00	01	11	10
00	1	0	0	0
01	0	1	x	1
11	x	x	x	x
10	x	x	x	x

10

b2	00	01	11	10
00	1	1	0	1
01	0	0	x	0
11	0	0	x	1
10	1	1	0	0

00

b1	00	01	11	10
00	0	0	1	1
01	0	0	x	1
11	0	1	x	0
10	0	1	0	1

01

b1	00	01	11	10
00	1	1	0	0
01	1	0	x	0
11	0	0	x	1
10	1	0	1	0

11

b1	00	01	11	10
00	1	0	1	0
01	1	0	x	0
11	x	x	x	x
10	x	x	x	x

10

b1	00	01	11	10
00	0	0	0	1
01	0	1	x	1
11	1	1	x	0
10	0	1	0	0

00

b0 \	00	01	11	10
00	0	1	1	0
01	0	1	x	0
11	1	0	x	0
10	1	0	0	1

01

b0 \	00	01	11	10
00	0	1	1	0
01	0	0	x	1
11	0	1	x	0
10	1	0	0	1

11

b0 \	00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	x	x	x	x
10	x	x	x	x

10

b0 \	00	01	11	10
00	0	1	0	0
01	1	0	x	1
11	0	1	x	0
10	1	0	1	0

5.4.5.7 Logic equations

uncorrected_sum_p2_b2 =
~b & c & ~e & ~f
+ c & ~d & ~e & ~f
+ a & b & d & ~f
+ b & ~c & e & ~f
+ ~b & ~c & ~e & f
+ a & c & e & f
+ b & c & e & f
+ ~a & ~b & c & ~d & ~f
+ a & ~b & ~c & d & e
+ ~a & ~b & ~c & ~d & f
+ ~a & ~c & ~d & ~e & f
+ ~a & b & c & d & f

uncorrected_sum_p2_b1 =
b & ~d & ~e & ~f
+ ~a & ~b & ~c & e
+ ~a & ~b & ~d & e
+ a & b & d & e
+ b & c & d & e
+ a & c & ~e & f
+ ~b & c & d & f
+ a & ~b & d & f
+ ~a & ~b & e & f
+ a & b & e & f

```
        + ~a & b & ~c & ~e & ~f
        + a & ~b & d & ~e & ~f
        + ~b & ~c & ~d & e & ~f
        + ~a & b & ~d & ~e & f

uncorrected_sum_p2_b0 =
        ~a & ~c & d & ~f
        + a & ~d & ~e & ~f
        + b & c & ~d & e
        + ~a & b & e & f
        + ~a & c & ~d & f
        + a & b & d & f
        + a & c & d & f
        + ~a & ~b & d & ~e & ~f
        + a & ~c & ~d & e & ~f
        + a & ~b & c & d & e
        + ~a & ~b & ~c & d & f
        + a & ~b & ~c & ~d & ~e & f
```

5.4.6 correct_mod_p1

5.4.6.1 Purpose

The correct_mod_p1 component computes the p1 segment (modulus 5) corrected sum from the corresponding intermediate sum. The carry_out value is used to determine what type of correction is necessary.

5.4.6.2 Inputs

carry_out: Carry value (2 bits).
uncorrected_sum_p1: Modulus 5 segment of the SDNR/RNS intermediate sum (3 bits).

5.4.6.3 Outputs

corrected_sum_p1: Modulus 5 segment of the SDNR/RNS corrected sum (3 bits).

5.4.6.4 Notes

For uncorrected_sum_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations. Likewise, for carry_out, the binary combination 11 does not exist.

```
        + ~a & b & ~c & ~e & ~f
        + a & ~b & d & ~e & ~f
        + ~b & ~c & ~d & e & ~f
        + ~a & b & ~d & ~e & f

uncorrected_sum_p2_b0 =
        ~a & ~c & d & ~f
        + a & ~d & ~e & ~f
        + b & c & ~d & e
        + ~a & b & e & f
        + ~a & c & ~d & f
        + a & b & d & f
        + a & c & d & f
        + ~a & ~b & d & ~e & ~f
        + a & ~c & ~d & e & ~f
        + a & ~b & c & d & e
        + ~a & ~b & ~c & d & f
        + a & ~b & ~c & ~d & ~e & f
```

5.4.6 correct_mod_p1

5.4.6.1 Purpose

The correct_mod_p1 component computes the p1 segment (modulus 5) corrected sum from the corresponding intermediate sum. The carry_out value is used to determine what type of correction is necessary.

5.4.6.2 Inputs

carry_out: Carry value (2 bits).
uncorrected_sum_p1: Modulus 5 segment of the SDNR/RNS intermediate sum (3 bits).

5.4.6.3 Outputs

corrected_sum_p1: Modulus 5 segment of the SDNR/RNS corrected sum (3 bits).

5.4.6.4 Notes

For uncorrected_sum_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations. Likewise, for carry_out, the binary combination 11 does not exist.

5.4.6.5 Truth table

carry_out	uncorrected_sum_p1	corrected_sum_p1 (b2 b1 b0)
00	000	010
01	000	000
10	000	011
00	001	011
01	001	001
10	001	100
00	010	100
01	010	010
10	010	000
00	011	000
01	011	011
10	011	001
00	100	001
01	100	100
10	100	010

5.4.6.6 Karnaugh maps

0

b2	00	01	11	10
00	0	0	0	1
01	0	x	x	x
11	1	x	x	x
10	0	0	0	0

1

b2	00	01	11	10
00	0	1	0	0
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

0

b1	00	01	11	10
00	1	1	0	0
01	0	x	x	x
11	0	x	x	x
10	0	0	1	1

1

b1	00	01	11	10
00	1	0	0	0
01	1	x	x	x
11	x	x	x	x
10	x	x	x	x

0

b0	00	01	11	10
00	0	1	0	0
01	1	x	x	x
11	0	x	x	x
10	0	1	1	0

1

b0	00	01	11	10
00	1	0	1	0
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

5.4.6.7 Logic equations

corrected_sum_p1_b2 =
c & d
+ a & ~b & e
+ ~a & b & ~d & ~e

corrected_sum_p1_b1 =
b & d
+ ~a & ~b & e
+ ~b & ~c & ~d & ~e

corrected_sum_p1_b0 =
a & d
+ a & ~b & ~e
+ a & b & e
+ c & ~d & ~e
+ ~a & ~b & ~c & ~d & e

5.4.7 correct_uod_p2

5.4.7.1 Purpose

The correct_mod_p2 component computes the p2 segment (modulus 7) corrected sum from the corresponding intermediate sum. The carry_out is used to determine what type of correction is necessary.

5.4.7.2 Inputs

carry_out: Carry value (2 bits).
uncorrected_sum_p2: Modulus 7 segment of the SDNR/RNS intermediate sum (3 bits).

5.4.7.3 Outputs

corrected_sum_p2: Modulus 7 segment of the SDNR/RNS corrected sum (3 bits).

5.4.7.4 Notes

For uncorrected_sum_p2, the binary combination 111 does not exist. Modulus 7 does not allow this combination. For carry_out, the binary combination 11 does not exist.

5.4.7.5 Truth table

carry_out	uncorrected_sum_p2	corrected_sum_p2 (b2 b1 b0)
00	000	100
01	000	000
10	000	011
00	001	101
01	001	001
10	001	100
00	010	110
01	010	010
10	010	101
00	011	000
01	011	011
10	011	110
00	100	001
01	100	100
10	100	000
00	101	010
01	101	101
10	101	001
00	110	011
01	110	110
10	110	010

5.4.7.6 Karnaugh maps

0

b2	00	01	11	10
00	1	1	0	1
01	0	0	x	0
11	1	1	x	1
10	0	0	0	0

1

b2	00	01	11	10
00	0	1	1	1
01	0	0	x	0
11	x	x	x	x
10	x	x	x	x

0

b1	00	01	11	10
00	0	0	0	1
01	0	1	x	1
11	0	0	x	1
10	0	0	1	1

1

b1	00	01	11	10
00	1	0	1	0
01	0	0	x	1
11	x	x	x	x
10	x	x	x	x

0

b0	00	01	11	10
00	0	1	0	0
01	1	0	x	1
11	0	1	x	0
10	0	1	1	0

1

b0	00	01	11	10
00	1	0	0	1
01	0	1	x	0
11	x	x	x	x
10	x	x	x	x

5.4.7.7 Logic equations

corrected_sum_p2_b2 =
c & d
+ a & b & e
+ ~a & ~c & ~d & ~e
+ ~b & ~c & ~d & ~e
+ a & ~c & ~d & e
+ b & ~c & ~d & e

corrected_sum_p2_b1 =
~a & b & ~e
+ b & d & ~e
+ a & b & e
+ b & c & e
+ a & c & ~d & ~e
+ ~a & ~b & ~c & ~d & e

corrected_sum_p2_b0 =
a & d
+ a & c & e
+ a & ~b & ~c & ~e
+ ~a & c & ~d & ~e
+ b & c & ~d & ~e
+ ~a & ~c & ~d & e

5.4.8 addc_mod_p1

5.4.8.1 Purpose

The addc_mod_p1 component adds the carry value, evaluated from the neighbouring SDNR/RNS digit adder, to the p1 segment (modulus 5) corrected sum.

5.4.8.2 Inputs

carry_in: Carry value from the neighbouring SDNR/RNS digit adder (2 bits).
corrected_sum_p1: Modulus 5 segment of the SDNR/RNS corrected sum (3 bits).

5.4.8.3 Outputs

sum_p1: Modulus 5 segment of the SDNR/RNS digit final sum (3 bits).

5.4.8.4 Notes

For corrected_sum_p1, the binary combinations 101, 110, and 111 do not exist. Modulus 5 does not allow these combinations. For carry_in, the binary combination 11 does not exist.

5.4.8.5 Truth table

carry_in	corrected_sum_p1	sum_p1 (b2 b1 b0)
00	000	100
01	000	000
10	000	001
00	001	000
01	001	001
10	001	010
00	010	001
01	010	010
10	010	011
00	011	010
01	011	011
10	011	100
00	100	011
01	100	100
10	100	000

5.4.8.6 Karnaugh maps

0

b2	00	01	11	10
00	1	0	0	0
01	0	x	x	x
11	1	x	x	x
10	0	0	0	0

1

b2	00	01	11	10
00	0	0	1	0
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

0

b1	00	01	11	10
00	0	0	1	0
01	1	x	x	x
11	0	x	x	x
10	0	0	1	1

1

b1	00	01	11	10
00	0	1	0	1
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

0

b0	00	01	11	10
00	0	0	0	1
01	1	x	x	x
11	0	x	x	x
10	0	1	1	0

1

b0	00	01	11	10
00	1	0	0	1
01	0	x	x	x
11	x	x	x	x
10	x	x	x	x

5.4.8.7 Logic equations

sum_p1_b2 =
c & d
+ a & b & e
+ ~a & ~b & ~c & ~d & ~e

sum_p1_b1 =

```
        b & d
        + ~a & b & e
        + a & ~b & e
        + a & b & ~e
        + c & ~d & ~e

sum_p1_b0 =
        a & d
        + ~a & b & ~d
        + ~a & b & e
        + c & ~d & ~e
        + ~a & ~c & ~d & e
```

5.4.9 addc_mod_p2

5.4.9.1 Purpose

The addc_mod_p2 component adds .the carry value, evaluated from tbe neighbouring SDNR/RNS digit adder, to the p2 segment (modulus 7) corrected sum.

5.4.9.2 Inputs

carry_in: Carry value from the neighbouring SDNR/RNS digit adder (2 bits).
corrected_sum_p2: Modulus 7 segment of the SDNR/RNS corrected sum (3 bits).

5.4.9.3 Outputs

sum_p2: Modulus 7 segment of the SDNR/RNS digit final sum (3 bits).

5.4.9.4 Notes

For corrected_sum_p2, the binary combination 111 does not exist. Modulus 7 does not allow this combination. For carry_in, the binary combination 11 does not exist.

5.4.9.5 Truth table

carry_in	corrected_sum_p2	sum_p2 (b2 b1 b0)
00	000	110
01	000	000
10	000	001
00	001	000
01	001	001
10	001	010
00	010	001
01	010	010
10	010	011
00	011	010
01	011	011
10	011	100
00	100	011
01	100	100
10	100	101
00	101	100
01	101	101
10	101	110
00	110	101
01	110	110
10	110	000

5.4.9.6 Karnaugh maps

0

b2	00	01	11	10
00	1	0	0	0
01	0	1	x	1
11	1	1	x	1
10	0	0	0	0

1

b2	00	01	11	10
00	0	0	1	0
01	1	1	x	0
11	x	x	x	x
10	x	x	x	x

0

b1	00	01	11	10
00	1	0	1	0
01	1	0	x	0
11	0	0	x	1
10	0	0	1	1

1

b1	00	01	11	10
00	0	1	0	1
01	0	1	x	0
11	x	x	x	x
10	x	x	x	x

0

b0	00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	0	1	x	0
10	0	1	1	0

1

b0	00	01	11	10
00	1	0	0	1
01	1	0	x	0
11	x	x	x	x
10	x	x	x	x

5.4.9.7 Logic equations

sum_p2_b2 =
a & c
+ c & d
+ a & b & e
+ ~b & c & e
+ b & c & ~e
+ ~a & ~b & ~c & ~d & ~e

sum_p2_b1 =
b & d
+ a & ~b & e
+ a & b & ~e
+ ~a & ~b & ~d & ~e
+ ~a & b & ~c & ~d & e

sum_p2_b0 =
a & d
+ ~a & ~b & e
+ ~a & ~c & ~d & e
+ ~a & b & ~d & ~e
+ ~a & c & ~d & ~e

5.5 VLSI considerations

The digit adder logic gates were based on dynamic logic. The digit adder itself was based on a four phase clock. By looking at Figure 37, it is easy to see that the digit adder could be broken up into five stages. By implementing a five stage digit adder, two cycles of the clock were required to evaluate a result. That is, the first four stages of the adder were executed in the first clock cycle, and the fifth stage could be executed in the second machine cycle. Furthermore, because the digit adder was broken up into stages, delay elements were required to maintain synchrony. Figure 38 shows the digit adder, with appropriately placed delay elements, split up into the five stages.

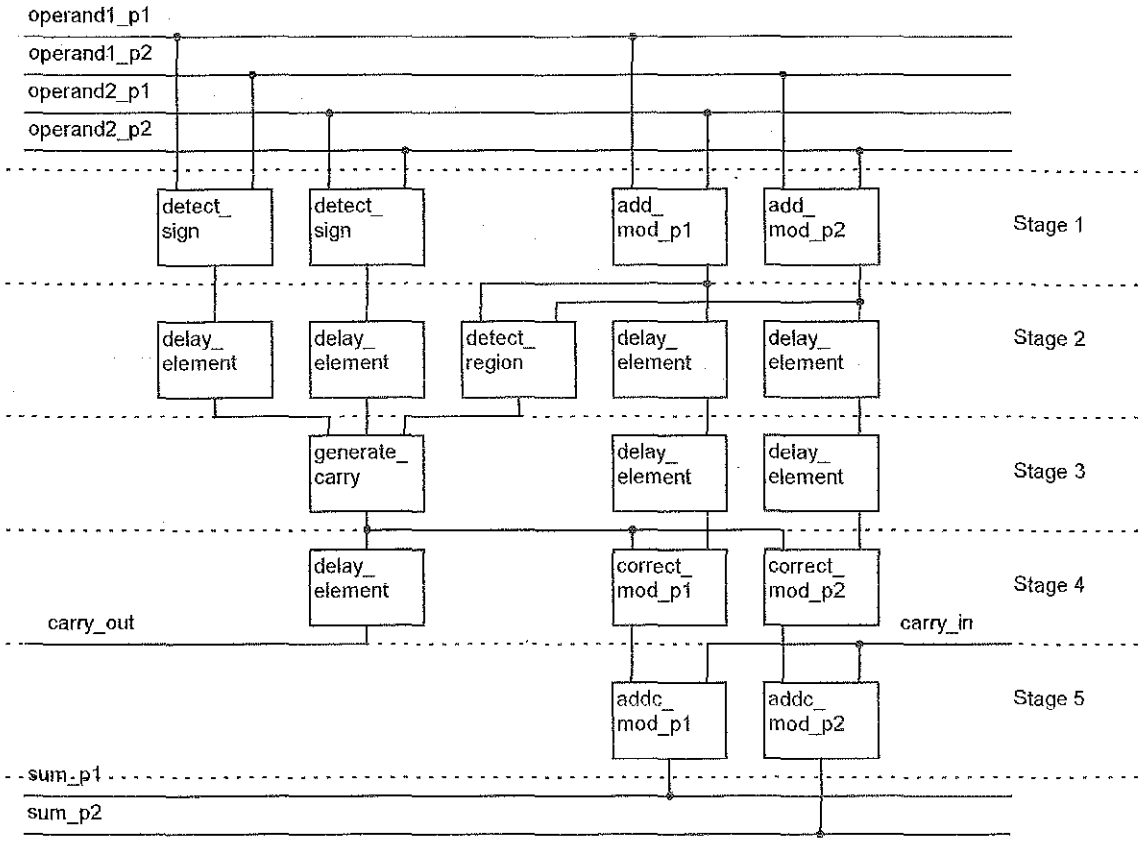

Figure 38: VLSI SDNR/RNS digit adder.

The basic delay element chosen for the SDNR/RNS digit adder, shown in Figure 38, was a D (delay) latch. A D latch was chosen because of its simplicity in operation. Figure 39 shows the logic diagram for a D latch.

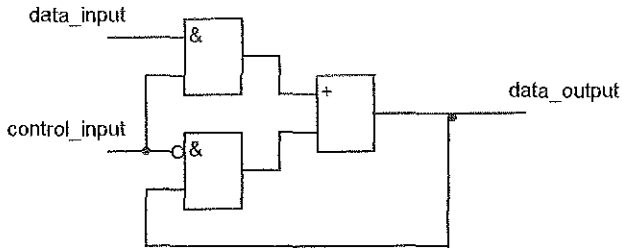

Figure 39: D latch.

5.6 Delay element design

5.6.1 delay_element

5.6.1.1 Purpose

The delay_element component holds the value of input data for the duration of the respective stage. A D-latch will be used for the delay_element. The D latch to be used is based on one described in Hayes (1993).

5.6.2 Inputs

data_input (D): The next data value to be stored in the D latch.
control_input (C): This is the enable input signal. In effect, this opens and closes the D latch feedback loop to allow new data to be entered or retained.

5.6.3 Outputs

data_output (Q [t] = D [t - t_1]): Delayed output of the input signal data_input. t_1 denotes the propagation delay within the D latch.

5.6.4 Notes

The three main parameters associated with a D latch are the setup time, hold time, and enable time. These parameters should not bear any influence in the implemented digit adder, because the delay of each stage, and hence the clock phase periods, are much greater than any of those three values.

5.6.5 Logic equations

$y^+ =$
D & C +
y & ~C

(The output Q is expressed as $Q = y^+$).

5.6.6 Logic equation refinements

Hayes (1993) points out that a static hazard exists for the D latch in Figure 39. Suppose that (D, C) = (1, 1), so that the latch is enabled and storing 1. The input combination applied to the OR gate is (1, 0), making Q = 1. Now let the enable signal C change to 0. This causes the OR gate's input to become (0, 1). If, due to differences in signal propagation delays, the upper signal to the OR gate changes from 1 to 0 before its lower input changes from 0 to 1, the OR gate briefly sees the input combination (0, 0), and may therefore produce a glitch in the form of a 0-pulse on Q. In fact, this spurious 0-pulse can become trapped in the latch's feedback loop, causing an incorrect transition to the reset state Q = 0. A spurious 0-pulse can also appear on Q when C changes from 0 to 1 with D = 1, but in this case Q returns to the correct state.

The static 1 hazard in this particular D latch is eliminated by adding an extra AND gate to the latch to generate the third, redundant term Dy of y^+. A D latch with this refinement is shown in Figure 40.

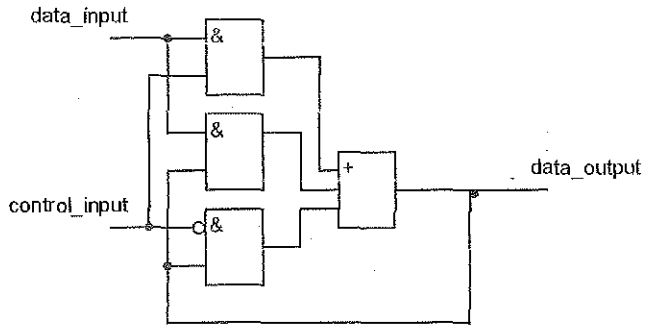

Figure 40: Hazard free D latch.

Now when the above glitch-inducing input condition occurs (C changes from 1 to 0), the fact that D and y are both 1 ensures that the output Dy of the new AND is 1. This 1 signal holds the output of the OR gate at a steady 1 while its other two input lines change in response to the changes in C (Hayes, 1993). Thus, a hazard free design, that deliberately includes a nonminimal AND-OR circuit, has been attained. The new logic equation is as follows:

$y^+ =$

D & C
+ D & y
+ y & ~C

The D latch logic equation has to be changed into a form which makes CMOS implementation easy. That is, for efficient implementation of the D latch, the corresponding logic equation must be in terms of NANDs and NORs. The transformation can be performed using Boolean algebra:

~y+ =

~(D & C
+ D & y
+ y & ~C)

~y+ =

~(D & C)
& ~(D & y)
& ~(y & ~C)

y+ =

~[~(D & C)
& ~(D & y)
& ~(y & ~C)]

Therefore, the D latch can be constructed using one not gate and four NAND gates.

5.7 Stick diagrams

Before the SDNR/RNS digit adder could be implemented in CMOS, stick diagrams of each type of logic gate were created. In short, stick diagrams depict the layout of a logic gate. Pucknell and Eshraghian (1994) explain stick diagrams in detail. For the adder, generalised stick diagrams were created for the domino logic gates (Figure 41), and the delay_elements (Figure 42).

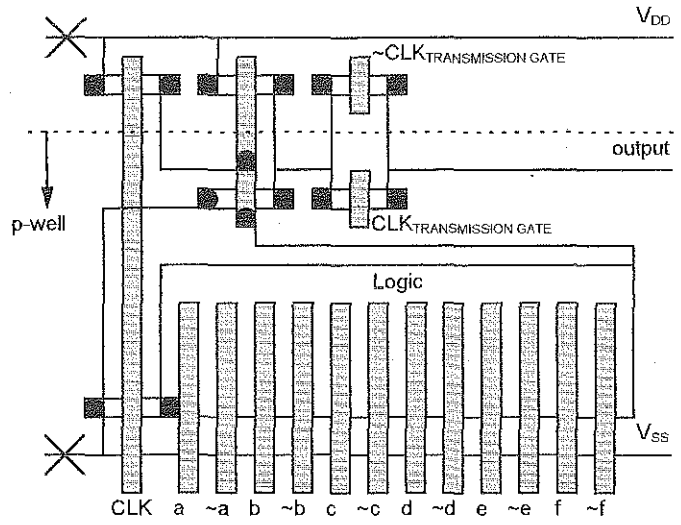

Figure 41: General stick diagram for domino logic gates.

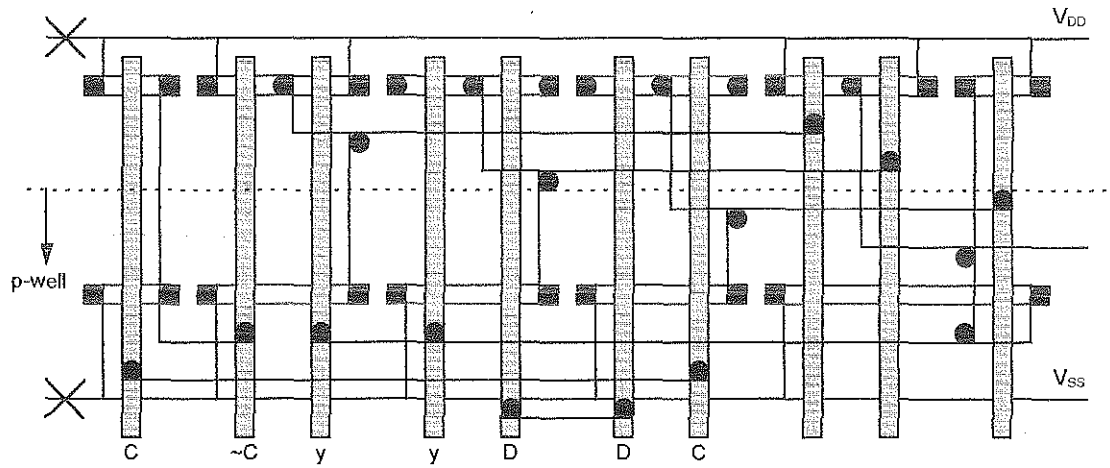

Figure 42: General stick diagram for delay_elements (D latches).

At the start of this project, the sign detector and the conventional to SDNR/RNS and SDNR/RNS to conventional number system converters, as well as the adder, were expected to be implemented. However, due to time constraints, only the SDNR/RNS digit adder was implemented using CMOS technology. Nevertheless, the designs of the sign detector and the converter systems have been included in this project report to illustrate the principles of operation of those particular logic circuits.

5.8 Sign detector design

The sign of a SDNR/RNS number can be determined by evaluating the sign of the most significant non-zero digit. From this statement, detecting the sign of a SDNR/RNS number sounds easy. However, it isn't quite that simple. Several logic components are required. In comparison, to detect the sign of a conventional binary number, only the most significant bit in the word need be stored.

To determine the sign of a SDNR/RNS number, the sign of each digit must first be determined. Then, a multiplexer, fed with the sign of each digit, must be used to select the sign of the most significant non-zero digit. The output of the multiplexer gives the sign of the number.

The functionality of the SDNR/RNS sign detector can be increased by designing it so that it can also detect zero. This would allow the sign detector circuit, with the inclusion of an array of digit adders, to be used as a magnitude comparator. A schematic of the sign detector is shown in Figure 43, and an illustration of a SDNR/RNS magnitude comparator, adapted from Kuczborski (1993), is shown in Figure 44.

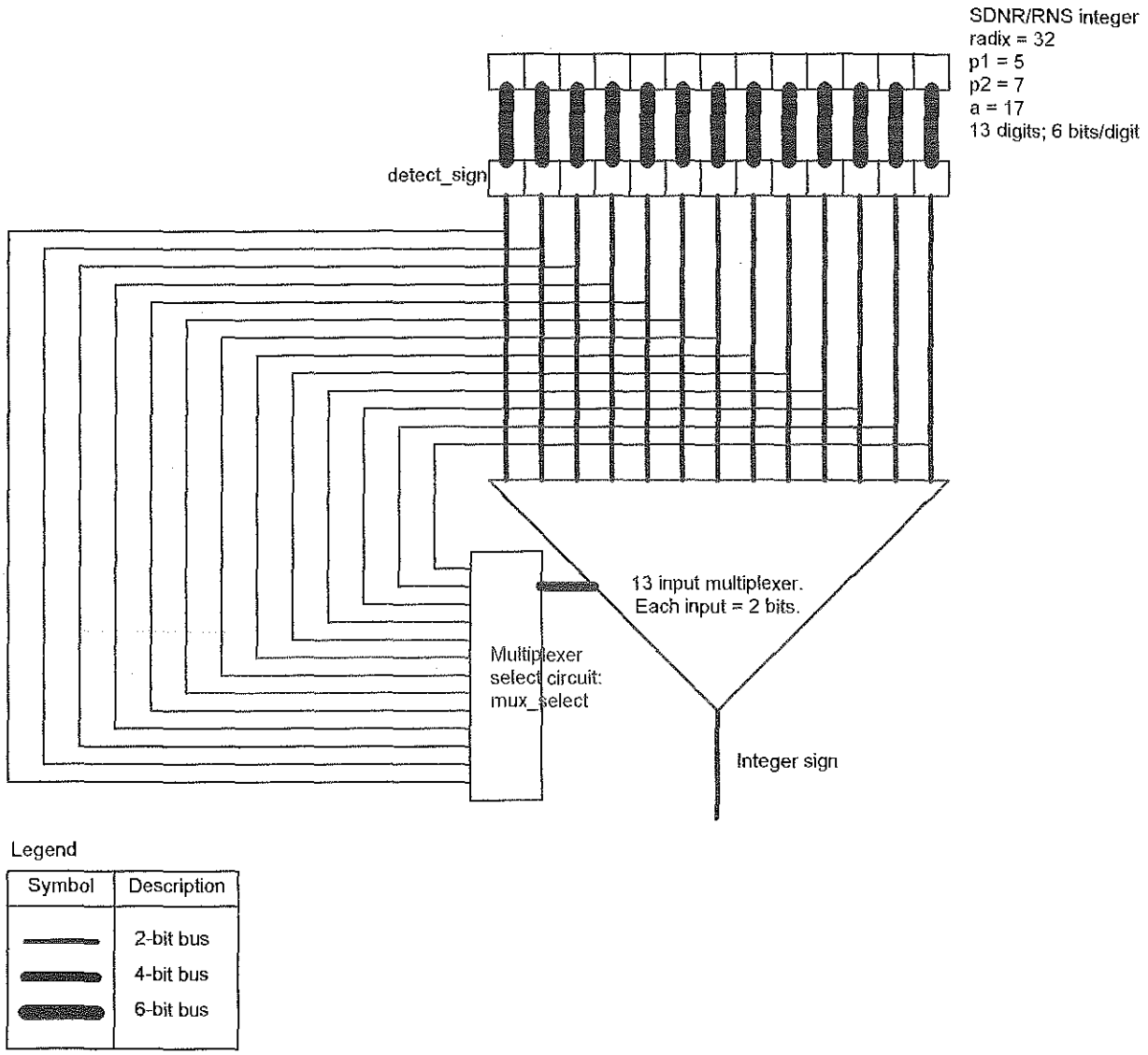

Figure 43: SDNR/RNS sign detector.

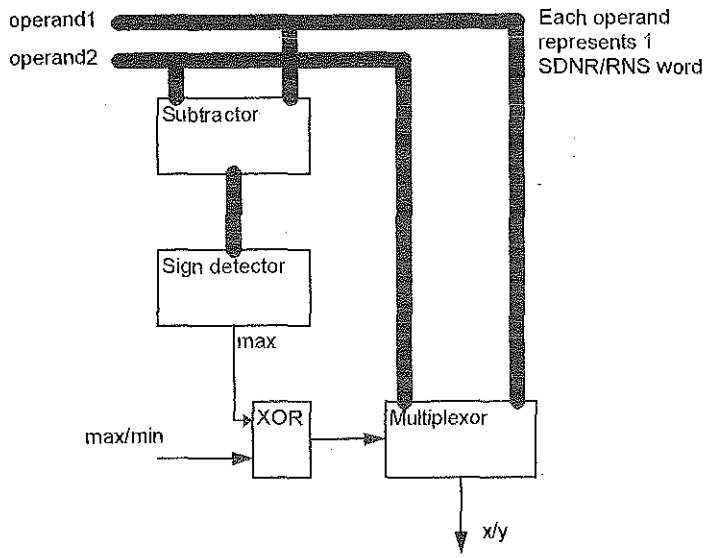

Figure 44: SDNR/RNS magnitude comparator.

Details of the SDNR/RNS sign detector components are presented in the following sections.

5.9 Sign detector component design

5.9.1 detect_sign

5.9.1.1 Purpose

The detect_sign component determines the sign of the input SDNR/RNS digit. As well as detecting sign, this component can detect zero.

5.9.1.2 Inputs

digit: The SDNR/RNS digit defined by <p1, p2> (6 bits).

5.9.1.3 Outputs

sign_digit: Sign of the SDNR/RNS digit (2 bits).

5.9.1.4 Notes

sign_digit_b1 identifies the unique representation of zero. The following algorithm is used to determine sign_digit_b1:

```
IF (the SDNR/RNS digit <p1, p2> is zero) THEN
        sign_digit_b1 = 0
ELSE
        sign_digit_b1 = 1
ENDIF
```

sign_digit_b0 identifies the sign of the number. The following algorithm is used to determine sign_digit_b0:

```
IF (SDNR/RNS digit <p1, p2> is positive) THEN
        sign_digit_b0 = 0
ELSE
        sign_digit_b0 = 1
ENDIF
```

The output bit sign_digit_b0 is equivalent to the output of the detect_sign module in the digit adder design.

5.9.1.5 Truth table

p1	p2	sign_ digit (b1 b0)
011	100	11
100	101	11
000	110	11
001	000	11
010	001	11
011	010	11
100	011	11
000	100	11
001	101	11
010	110	11
011	000	11
100	001	11
000	010	11
001	011	11
010	100	11
011	101	11
100	110	11
000	000	00
001	001	10
010	010	10
011	011	10
100	100	10
000	101	10
001	110	10
010	000	10
011	001	10
100	010	10
000	011	10
001	100	10
010	101	10
011	110	10
100	000	10
000	001	10
001	010	10
010	011	10

5.9.1.6 Karnaugh maps

00

b1 \ 00	01	11	10
00	0	1	1	1
01	1	1	x	1
11	1	1	x	1
10	1	1	1	1

01

b1 \ 00	01	11	10
00	1	1	1	1
01	1	1	x	1
11	1	1	x	1
10	1	1	1	1

11

b1 \ 00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b1 \ 00	01	11	10
00	1	1	1	1
01	1	1	x	1
11	x	x	x	x
10	x	x	x	x

00

b0 \ 00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	0	1	x	0
10	1	0	1	0

01

b0 \ 00	01	11	10
00	0	1	0	0
01	1	0	x	1
11	1	1	x	0
10	1	0	0	1

11

b0 \ 00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b0 \ 00	01	11	10
00	0	1	1	0
01	0	1	x	1
11	x	x	x	x
10	x	x	x	x

5.9.1.7 Logic equations

sign_digit_b1 =
~a & ~b & ~c & ~d & ~e & ~f
= a + b + c + d + e + f

sign_digit_b0 =
~a & c & ~d & ~f
+ ~a & ~b & ~c & d & ~e & ~f
+ a & c & d & ~f
+ a & b & d & ~e & ~f
+ ~a & b & ~d & ~e & ~f
+ ~a & ~b & c & e
+ ~a & ~c & d & e
+ a & ~b & ~c & ~d & e
+ a & f
+ b & c & f

5.9.2 mux_select

5.9.2.1 Purpose

The mux_select component contains the selection logic for the multiplexer.

5.9.2.2 Inputs

not_zero_digit: This is an array of 13 bits which determine the zero signature pattern of the SDNR/RNS number. Each one bit element (d12, d11, ..., d01, d00) in this array is equivalent to the sign_digit_b1 output of the respective detect_sign gate. The most significant bit in this array (d12) represents the zero status of the most significant SDNR/RNS digit. Likewise, the least significant bit in this array (d00) represents the zero status of the least significant SDNR/RNS digit.

5.9.2.3 Outputs

mux_select_code: A code representing the sign_digit bit pair to be selected to represent the zero and sign status of the SDNR/RNS number (4 bits).

5.9.2.4 Notes

The following algorithm is used to determine the output of the mux_select component:

CASEOF mux_select_code
d00:
mux_select_code = 0000
d01:
mux_select_code = 0001
d02:
mux_select_code = 0010
d03:
mux_select_code = 0011

```
        d04:
                mux_select_code = 0100
        d05:
                mux_select_code = 0101
        d06:
                mux_select_code = 0110
        d07:
                mux_select_code = 0111
        d08:
                mux_select_code = 1000
        d09:
                mux_select_code = 1001
        d10:
                mux_select_code = 1010
        d11:
                mux_select_code = 1011
        d12:
                mux_select_code = 1100
ENDCASE
```

5.9.2.5 Truth table

d12	d11	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00	mux_select_code (b3 b2 b1 b0)
1	x	x	x	x	x	x	x	x	x	x	x	x	1100
0	1	x	x	x	x	x	x	x	x	x	x	x	1011
0	0	1	x	x	x	x	x	x	x	x	x	x	1010
0	0	0	1	x	x	x	x	x	x	x	x	x	1001
0	0	0	0	1	x	x	x	x	x	x	x	x	1000
0	0	0	0	0	1	x	x	x	x	x	x	x	0111
0	0	0	0	0	0	1	x	x	x	x	x	x	0110
0	0	0	0	0	0	0	1	x	x	x	x	x	0101
0	0	0	0	0	0	0	0	1	x	x	x	x	0100
0	0	0	0	0	0	0	0	0	1	x	x	x	0011
0	0	0	0	0	0	0	0	0	0	1	x	x	0010
0	0	0	0	0	0	0	0	0	0	0	1	x	0001
0	0	0	0	0	0	0	0	0	0	0	0	1	0000
0	0	0	0	0	0	0	0	0	0	0	0	0	0000

5.9.2.6 Logic equations

mux_select_code_b3 =
d12
+ ~d12 & d11
+ ~d12 & ~d11 & d10
+ ~d12 & ~d11 & ~d10 & d09
+ ~d12 & ~d11 & ~d10 & ~d09 & d08

= d12 + ~d12 & (d11 + ~d11 & (d10 + ~d10 & (d09 + ~d09 & d08)))

mux_select_code_b2 =
d12
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & d07
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & d06
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & d05
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & ~d05 & d04

= d12 + ~d12 & (~d11 & (~d10 & (~d09 & (~d08 & (d07 + ~d08 & (~d07 & (d06 + ~d06 & (d05+ ~d05 & d04)))))))

mux_select_code_b1 =
~d12 & d11
+ ~d12 & ~d11 & d10
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & d07
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & d06
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & ~d05 & ~d04 & d03
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & ~d05 & ~d04 & ~d03 & d02

= ~d12 & (d11+ ~d11 & (d10 + ~d10 & (~d09 & (~d08 & (d07 + ~d07 & (d06 + ~d06 & (~d05 & (~d04 & (d03 + ~d03 & d02)))))))))

mux_select_code_b0 =
~d12 & d11
+ ~d12 & ~d11 & ~d10 & d09
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & d07
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & d05
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & ~d05 & ~d04 & d03
+ ~d12 & ~d11 & ~d10 & ~d09 & ~d08 & ~d07 & ~d06 & ~d05 & ~d04 & ~d03 & ~d02 & d01

= ~d12 & (d11 + ~d11 & (~d10 & (d09 + ~d09 & (~d08 & (d07 + ~d07 & (~d06 & (d05 + ~d05 & (~d04 & (d03 + ~d03 & ~d02 & d01)))))))))

5.10 Converter designs

There are two number system conversion circuits required when dealing witb a nonconventional number system. In this case, the nonconventional number system is the SDNR/RNS data representation. The circuits required for conversion are as follows:

1. Conventional to SDNR/RNS number system conversion.
2. SDNR/RNS to conventional number system conversion.

The following sections detail the logic design of parallel conversion systems. Note, however, that the conversion circuit design was simplified by only allowing the conversion of unsigned conventional integers into their nonconventional equivalent. The converters in this project report demonstrate the principles of conversion, even though not allowing signed integer (for example, two's complement) conversion limits the application of such logic.

5.10.1 Conventional to SDNR/RNS number system conversion

The conventional to SDNR number system conversion algorithm presented in section 3.1.4.4-Conventional number systems to SDNR conversion, bears a striking resemblance to the SDNR/RNS digit adder algorithm in section 3.1.7.1-SDNR/RNS addition. The logic realisation of the conversion algorithm is also similar, and is shown in Figure 45. The logic circuit, like the adder, converts number on a digit by digit basis. An array of 13 converters would be required for the chosen radix 32 SDNR/RNS configuration. Some of the logic components from the SDNR/RNS digit adder can be used in the converter logic. These components are as follows:

1. correct_mod_p1.
2. correct_mod_p2.
3. addc_mod_p1.
4. addc_mod_p2.

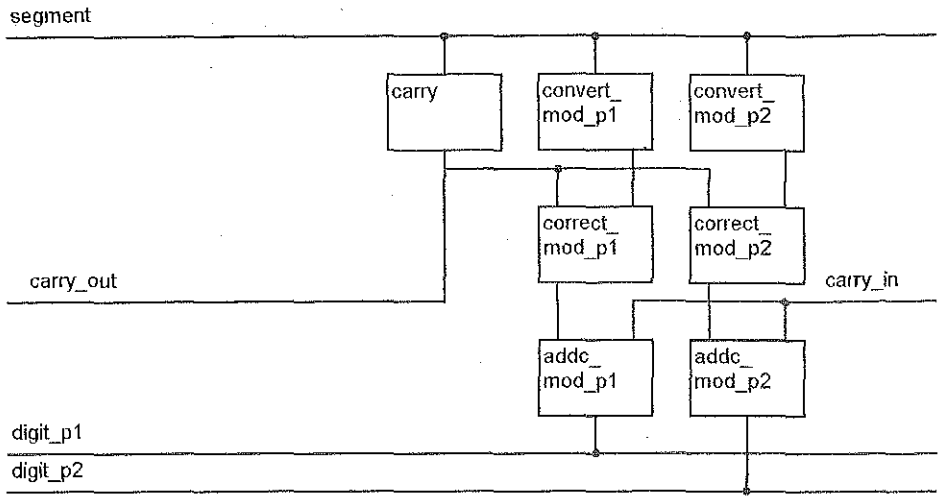

Figure 45: Conventional to SDNR/RNS number system converter.

Details of the design for the carry, convert_mod_p1, and convert_mod_p2 components are presented in the following sections.

5.10.2 Component design

5.10.2.1 carry

5.10.2.1.1 Purpose

The carry component determines the carry_out value. The output of this component is based upon the selected threshold value.

5.10.2.1.2 Inputs

segment: A 5-bit segment of the unsigned conventional binary number. The SDNR/RNS configuration is based on radix 32. Therefore, the conversion process can be completed by grouping the conventional number in clusters of ($\log_2(32)$ =) 5 bits.

5.10.2.1.3 Outputs

carry_out: The carry value to pass onto the neighbouring digit converter (2 bits).

5.10.2.1.4 Notes

The following algorithm is used to determine the output of the carry component:

```
IF segment <= t THEN
        carry_out = 01
ELSE
        carry_out = 00
ENDIF
```

where t = threshold value (refer to section 3.1.4.4-Conventional number systems to SDNR conversion, for information on the threshold value).

5.10.2.1.5 Truth table

segment	carry_out (b1 $\overline{b0}$)
00000	01
00001	01
00010	01
00011	01
00100	01
00101	01
00110	01
00111	01
01000	01
01001	01
01010	01
01011	01
01100	01
01101	01
01110	01
01111	01
10000	01
10001	00
10010	00
10011	00
10100	00
10101	00
10110	00
10111	00
11000	00
11001	00
11010	00
11011	00
11100	00
11101	00
11110	00
11111	00

5.10.2.1.6 Karnaugh maps

0

b1 \	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

1

b1 \	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

0

b0	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

1

b0	00	01	11	10
00	1	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

5.10.2.1.7 Logic equations

carry_out_b1 =
0

carry_out_b0 =
~e
+ ~a & ~b & ~c & ~d & e

5.10.2.2 convert_mod_p1

5.10.2.2.1 Purpose

The convert_mod_p1 component converts a conventional binary 5-bit segment into the equivalent p1 (modulus 5) modulus.

5.10.2.2.2 Inputs

segment: A segment of the unsigned conventional binary number (5 bits).

5.10.2.2.3 Outputs

uncorrected_convert_p1: The uncorrected modulus 5 element of the equivalent SDNR/RNS digit (3 bits).

5.10.2.2.4 Notes

The following equation is used to determine the output of the convert_mod_p1 component:

uncorrected_convert_p1 = segment MOD 5

5.10.2.2.5 Truth table

segment	uncorrected_convert_p1 (b2 b1 b0)
00000	000
00001	001
00010	010
00011	011
00100	100
00101	000
00110	001
00111	010
01000	011
01001	100
01010	000
01011	001
01100	010
01101	011
01110	100
01111	000
10000	001
10001	010
10010	011
10011	100
10100	000
10101	001
10110	010
10111	011
11000	100
11001	000
11010	001
11011	010
11100	011
11101	100
11110	000
11111	001

5.10.2.2.6 Karnaugh maps

0

b2	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	0	0	1
10	0	1	0	0

1

b2	00	01	11	10
00	0	0	1	0
01	0	0	0	0
11	0	1	0	0
10	1	0	0	0

0

b1	00	01	11	10
00	0	0	1	1
01	0	0	1	0
11	1	1	0	0
10	1	0	0	0

1

b1	00	01	11	10
00	0	1	0	1
01	0	0	1	1
11	1	0	0	0
10	0	0	1	0

0

b0	00	01	11	10
00	0	1	1	0
01	0	0	0	1
11	0	1	0	0
10	1	0	1	0

1

b0	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	1	0	1	0
10	0	0	0	1

5.10.2.2.7 Logic equations

uncorrected_convert_p1_b2 =
~a & ~b & ~c & d & e
+ ~a & ~b & c & ~d & ~e
+ ~a & b & c & d & ~e
+ a & ~b & ~c & d & ~e
+ a & ~b & c & d & e
+ a & b & ~c & ~d & e

uncorrected_convert_p1_b1 =
~a & ~b & c & d
+ ~a & ~b & d & ~e
+ ~a & b & ~d & e
+ ~b & c & d & ~e
+ a & b & ~d & ~e
+ b & ~c & ~d & ~e
+ b & c & ~d & e
+ a & ~b & ~c & ~d & e
+ a & b & ~c & d & e

uncorrected_convert_p1_b0 =
~a & ~c & ~d & e
+ ~a & b & ~c & e
+ a & ~c & ~d & ~e
+ a & b & ~c & ~e
+ a & b & c & e

+ a & c & ~d & e
+ ~a & ~b & ~c & d & ~e
+ ~a & b & c & ~d & ~e
+ a & ~b & c & d & ~e
+ ~a & ~b & c & d & e

5.10.2.3 convert_mod_p2

5.10.2.3.1 Purpose

The convert_mod_p2 component converts a conventional binary 5-bit segment into the equivalent p2 (modulus 7) modulus.

5.10.2.3.2 Inputs

segment: A segment of the unsigned conventional binary number (5 bits).

5.10.2.3.3 Outputs

uncorrected_convert_p2: The uncorrected modulus 7 element of the equivalent SDNR/RNS digit (3 bits).

5.10.2.3.4 Notes

The following equation is used to determine the output of the convert_mod_p2 component:

uncorrected_convert_p2 = segment MOD 7

5.10.2.3.5 Truth table

segment	uncorrected_convert_p2 (b2 b1 b0)
00000	000
00001	001
00010	010
00011	011
00100	100
00101	101
00110	110
00111	000
01000	001
01001	010
01010	011
01011	100
01100	101
01101	110
01110	000
01111	001
10000	010
10001	011
10010	100
10011	101
10100	110
10101	000
10110	001
10111	010
11000	011
11001	100
11010	101
11011	110
11100	000
11101	001
11110	010
11111	011

5.10.2.3.6 Karnaugh maps

0

b2 \	00	01	11	10
00	0	0	0	0
01	1	1	0	1
11	1	1	0	0
10	0	0	1	0

1

b2 \	00	01	11	10
00	0	0	1	1
01	1	0	0	0
11	0	0	0	0
10	0	1	1	1

0

b1	00	01	11	10
00	0	0	1	1
01	0	0	0	1
11	0	1	0	0
10	0	1	0	1

1

b1	00	01	11	10
00	1	1	0	0
01	1	0	1	0
11	0	0	1	1
10	1	0	1	0

0

b0	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	1	0	1	0
10	1	0	0	1

1

b0	00	01	11	10
00	0	1	1	0
01	0	0	0	1
11	0	1	1	0
10	1	0	0	1

5.10.2.3.7 Logic equations

uncorrected_convert_p2_b2 =
~b & c & ~e
+ b & ~c & e
+ ~a & ~b & c & ~d
+ ~a & c & ~d & ~e
+ a & b & ~c & d
+ a & ~c & d & e

uncorrected_convert_p2_b1 =
~a & b & ~c & ~e
+ ~a & b & ~d & ~e
+ a & ~b & d & ~e
+ b & ~c & ~d & ~e
+ ~a & ~b & ~c & e
+ ~a & ~b & ~d & e
+ ~b & ~c & ~d & e
+ a & b & c & e
+ a & b & d & e
+ b & c & d & e

uncorrected_convert_p2_b0 =
~a & ~c & d
+ a & ~c & ~d
+ ~a & ~b & d & ~e
+ a & ~b & ~d & ~e

+ a & b & c & d
+ a & c & d & e
+ ~a & b & c & ~d & e

5.10.3 SDNR/RNS to conventional number system conversion

A 64-bit borrow lookahead subtracter is used to convert an SDNR/RNS number back to conventional notation. It is a borrow lookahead subtracter because the conventional to SDNR/RNS conversion performs unsigned conversion. Rajashekhara and Nale (1990) report that on-line conversion from signed digit to radix complement representation is possible by using converter hardware which consists of borrow lookback and decrementer units. Hayes (1993) describes the similarity between a borrow adder and a carry adder.

Figure 46 shows the logic components required to convert back to unsigned conventional notation. The detect_sign logic gate is the same as that used in the SDNR/RNS digit adder. Each SDNR/RNS digit is converted to a conventional segment on a digit by digit basis. An array of 13 of these digit converters would be required to change a SDNR/RNS number, based on the chosen configuration, back into conventional notation. However, the algorithm required for conversion to the conventional domain is not as modular as that for conversion into the SDNR/RNS. The borrow_lookahead_subtracter component is required by all digit converters, and thus modularity suffers.

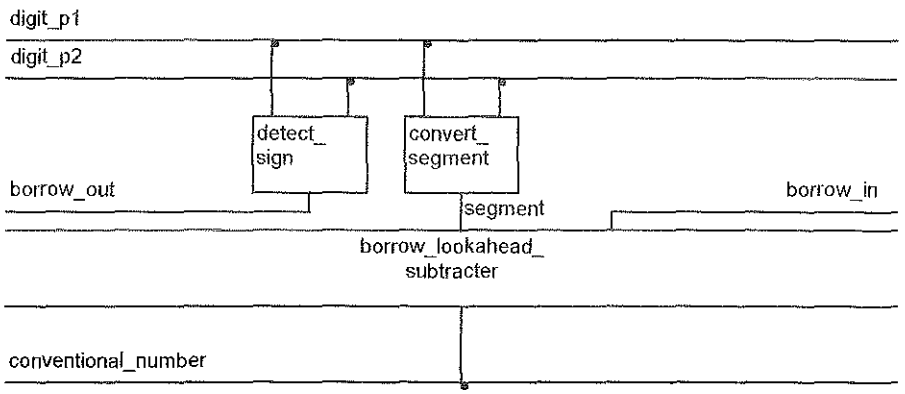

Figure 46: SDNR/RNS to conventional number system converter.

Design of the convert_segment and borrow_lookahead_subtracter components are detailed in the following sections. The borrow_lookahead_subtracter is based on a set of logic equations derived by Hayes (1993).

5.10.4 Component design

5.10.4.1 convert_segment

5.10.4.1.1 Purpose

The convert_segment component converts the SDNR/RNS digit into an uncorrected conventional binary 5-bit segment.

5.10.4.1.2 Inputs

digit_p1: Modulus 5 element of the SDNR/RNS digit to be converted (3 bits).

digit_p2: Modulus 7 element of the SDNR/RNS digit to be converted (3 bits).

5.10.4.1.3 Outputs

segment: Uncorrected conventional segment value (5 bits).

5.10.4.1.4 Truth table

digit_p1	digit_p2	segment (b4 b3 b2 b1 b0)
011	100	01111
100	101	10000
000	110	10001
001	000	10010
010	001	10011
011	010	10100
100	011	10101
000	100	10110
001	101	10111
010	110	11000
011	000	11001
100	001	11010
000	010	11011
001	011	11100
010	100	11101
011	101	11110
100	110	11111
000	000	00000
001	001	00001
010	010	00010
011	011	00011
100	100	00100
000	101	00101
001	110	00110
010	000	00111
011	001	01000
100	010	01001
000	011	01010
001	100	01011
010	101	01100
011	110	01101
100	000	01110
000	001	01111
001	010	10000
010	011	10001

5.10.4.1.5 Karnaugh maps

00

b4	00	01	11	10
00	0	0	0	1
01	1	0	x	1
11	0	1	x	0
10	1	0	1	1

01

b4	00	01	11	10
00	0	1	1	0
01	1	0	x	1
11	0	1	x	0
10	1	0	0	1

11

b4	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b4	00	01	11	10
00	0	1	1	0
01	0	1	x	1
11	x	x	x	x
10	x	x	x	x

00

b3	00	01	11	10
00	0	1	1	1
01	0	0	x	0
11	1	0	x	0
10	0	0	1	0

01

b3	00	01	11	10
00	0	0	0	0
01	1	1	x	1
11	1	1	x	1
10	1	1	0	0

11

b3	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b3	00	01	11	10
00	1	1	0	1
01	0	0	x	1
11	x	x	x	x
10	x	x	x	x

00

b2	00	01	11	10
00	0	1	0	0
01	1	1	x	0
11	0	1	x	1
10	0	0	1	0

01

b2	00	01	11	10
00	1	0	0	0
01	1	1	x	0
11	1	1	x	1
10	0	0	0	1

11

b2	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b2	00	01	11	10
00	1	0	1	0
01	1	0	x	1
11	x	x	x	x
10	x	x	x	x

00

b1	00	01	11	10
00	0	1	1	1
01	1	0	x	0
11	1	1	x	1
10	1	0	0	0

01

b1	00	01	11	10
00	1	1	0	1
01	0	0	x	0
11	1	1	x	0
10	0	0	1	0

11

b1	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b1	00	01	11	10
00	1	1	0	0
01	0	0	x	1
11	x	x	x	x
10	x	x	x	x

00

b0 \	00	01	11	10
00	0	1	0	1
01	0	1	x	1
11	1	1	x	0
10	0	1	0	0

01

b0 \	00	01	11	10
00	1	1	1	0
01	1	0	x	0
11	1	0	x	1
10	1	0	1	0

11

b0 \	00	01	11	10
00	x	x	x	x
01	x	x	x	x
11	x	x	x	x
10	x	x	x	x

10

b0 \	00	01	11	10
00	0	0	1	1
01	0	0	x	1
11	x	x	x	x
10	x	x	x	x

5.10.4.1.6 Logic equations

segment_b4 =
~a & c & ~d & ~f
+ ~a & ~c & d
+ a & c & d
+ a & b & d & ~e
+ ~a & b & ~c & ~e & ~f
+ a & ~c & ~d & e
+ a & f
+ b & c & f

segment_b3 =
~a & ~b & c & d
+ a & ~c & ~d & ~e & ~f
+ a & b & ~e & ~f
+ b & ~c & ~d & ~e & ~f
+ c & e
+ ~b & d & e
+ ~a & b & f
+ ~b & ~c & ~d & f

segment_b2 =
~b & c & ~d & ~f
+ a & ~b & ~d & ~e & ~f
+ a & c & ~f
+ a & b & d & ~e
+ b & c & d
+ ~a & ~b & ~d & e

```
        + ~b & c & e
        + ~a & b & d & e
        + ~a & ~b & f
        + a & b & f
        + b & c & f

segment_b1 =
        ~a & ~b & c & ~e & ~f
        + ~a & ~b & d & ~e
        + ~b & c & d
        + c & d & ~e
        + a & ~c & ~d & ~e & ~f
        + b & ~c & ~d & ~e & ~f
        + ~b & ~c & ~d & e
        + ~a & ~c & ~d & e
        + a & b & d & e
        + b & c & f
        + ~b & ~c & ~d & ~e & f

segment_b0 =
        ~b & c & d & ~e
        + a & ~b & ~e & ~f
        + ~a & b & ~d & ~e
        + ~a & ~b & e
        + ~b & ~c & ~d & e
        + a & b & e
        + b & c & d & e
        + b & f
```

5.10.4.2 borrow_lookahead_subtracter

5.10.4.2.1 Purpose

The borrow_lookahead_subtracter component subtracts all uncorrected segments and all borrows so that the corrected unsigned conventional integer is obtained.

5.10.4.2.2 Inputs

operand1: All evaluated segments (64 bits).
operand2: All evaluated borrows (64 bits).

5.10.4.2.3 Outputs

conventional_number: The unsigned conventional binary integer (64 bits).

5.10.4.2.4 Logic equations

Borrow adder equations:

$d_i = x_i \oplus b_i \oplus b_{i-1}$
$b_i = \sim x_i y_i + \sim x_i b_{i-1} + y_i b_{i-1}$

Borrow lookahead subtracter equations:

$b_i = g_i + p_i b_{i-1}$

$g_i = \sim x_i y_i$
$p_i = \sim x_i + y_i$

where d_i = ith difference bit.
x_i = ith operand1 bit.
y_i = ith operand2 bit.
b_i = ith borrow bit.
g_i = ith bit generate function.
p_i = ith bit propagate function.

6. Testing

The testing stage involved verifying the logic of the SDNR/RNS number system converters, digit adder, and sign detector. Functions written in C were used to represent each logic gate in the respective system. For example, the detect_sign gate in the digit adder was coded as the detect_sign function in the C software simulation program. Section 10.1-Appendix A: Software simulation system lists the source code for the SDNR/RNS software simulation system. All functions were unit and system tested to determine mistakes made during the design phase of the project. There are two main programs in this software simulation system. The first one tests the SDNR/RNS number system converters and the digit adder, while the second one simulates the logic for the sign detector.

6.1 The SDNR/RNS number system converters and the digit adder

6.1.1 Unit testing

Unit testing involved creating an algorithm which tested every single logic state of the number system conversion and the digit adder functions. This was done by writing all input truth tables to file, and then reading them into the appropriate function. The outputs from the functions were analysed against the expected truth table outputs, and errors were investigated and fixed. The unit testing algorithm could be described as follows:

1. Open the function's truth table file.
2. LOOP while not end of file
   a) From file, read in next truth table entry.
   b) Execute function, with truth table entry as input, and a result as output.
   c) Display function result.
3. ENDLOOP
4. Close the file.

This process was repeated until all software simulation functions gave the correct truth table outputs.

6.1.2 System testing

The number system conversion and digit adder functions were tested at the system level by implementing the algorithms discussed in sections 3.1.4.4-Conventional number systems to SDNR conversion, 3.1.4.5-SDNR to conventional number systems conversion, and 3.1.7.1-SDNR/RNS addition. The system level test program can be described by the following high level algorithm:

1. Prompt user for two conventional radix 32 operands (each digit can take on the value 0 to 31).
2. Convert conventional radix 32 operands to radix 32 SDNR/RNS representation.
3. Add the two operands using an array of 13 digit adders (remember 13 digit adders to satisfy the range for adding two 64-bit integers).
4. Convert the radix 32 SDNR/RNS sum to conventional radix 32 representation.
5. Display the conventional radix 32 sum.

There were two kinds of system level tests, and they were single digit additions, and multiple digit additions.

6.1.2.1 Single digit additions

The single digit addition test plan was devised to verify the addition of various radix 32 conventional numbers. The test plan is shown in Table 22. Note that the test plan lists the SDNR/RNS digits in terms of SDNR representation. This allowed for easy visual verification of additions. Some entries in Table 22 tested the carry output logic of the digit adder.

Table 22: Single digit addition test plan.

operand1	operand2	carry_in	sum	Comment
22	15	0	(1) (5)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.
4	11	0	(15)	
12	17	0	(29)	
10	6	0	(16)	
7	10	0	(17)	
26	22	0	(1) (16)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.
22	25	0	(1) (15)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.
15	15	0	(30)	
17	17	0	(1) (2)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.

6.1.2.2 Multiple digit additions

The multiple digit addition test plan was devised to verify the addition of various radix 32 conventional numbers, which had word lengths greater than one digit. The test plan is shown in Table 23. This test plan focused on making sure that each digit adder could add the operand digits as well as handle carry in values and produce carry out values correctly.

Table 23: Multiple digit addition test plan.

operand1 (MSD ... LSD)	operand2 (MSD ... LSD)	sum	Comment
(3) (20)	(2) (27)	(6) (15)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.
(4) (17)	(9) (16)	(14) (1)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.
(9) (13)	(30) (10)	(1) (7) (23)	Tested local carry propagation. Carry from least to the next significant digit adder = 1. Carry from the second least to third least significant digit adder = 1.
(5) (18) (4) (30) (31) (14) (1) (0) (25) (7) (19) (21) (3)	(2) (9) (31) (31) (17) (6) (29) (24) (22) (0) (0) (13) (11)	(7) (28) (4) (30) (16) (20) (30) (25) (15) (7) (20) (2) (14)	Tested addition of two 13 digit operands.

6.2 The SDNR/RNS number system converters and the sign detector

6.2.1 Unit testing

Unit testing in this case involved creating an algorithm which tested every single logic state of the sign detector functions. The algorithm used for unit testing was the same one listed in section 6.1.1- Unit testing.

6.2.2 System testing

System level testing involved implementing the sign detector algorithm implied by Figure 43. The algorithm can be described as follows:

1. Prompt user for a conventional radix 32 operand (each digit can take on the value 0 to 31).
2. Convert conventional radix 32 operand to radix 32 SDNR/RNS representation.
3. Detect sign of all 13 digits. Each digit is evaluated as either negative, positive, or zero.
4. Select the MSD.
5. Display the sign of the MSD.

The sign detector test plan is listed in Table 24. The aim of the plan was to test every output of the multiplexer selection logic. That is, every one of the 13 digit positions in the SDNR/RNS word were set as the MSD, to see whether the sign of the operand could be determined.

Table 24: Sign detector test plan.

operand	sign
(0)	00
(28)	10
(15) (1)	10
(28) (20) (27)	10
(4) (22) (30) (26)	10
(5) (8) (0) (13) (16)	10
(23) (25) (30) (3) (19) (22)	10
(22) (19) (14) (12) (17) (23) (30)	10
(20) (24) (29) (1) (17) (20) (10) (9)	10
(10) (19) (26) (3) (8) (18) (16) (26) (10)	10
(11) (31) (20) (24) (15) (15) (4) (5) (13) (21)	10
(12) (31) (28) (23) (6) (15) (22) (19) (29) (5) (21)	10
(2) (27) (27) (13) (18) (24) (15) (4) (14) (0) (22) (12)	10
(13) (16) (5) (24) (18) (8) (10) (22) (14) (17) (10) (3) (9)	10

7. Implementation

The digit adder was the only arithmetic subsystem to be implemented. The CMOS realisation of the radix 32 SDNR/RNS digit adder is shown in Figure 47 (the digit adder has been rotated anticlockwise 90°, so that it can fit on the page). The CAD software package used for the layout of the adder was called Magic (version 6.4.5). The design rule checker used was based on SCMOS (version 1.0.0).

The digit adder had a width of 1250λ, and a height of 818λ. In total, the digit adder contained 1282 transistors (922 n-transistors, and 360 p-transistors).

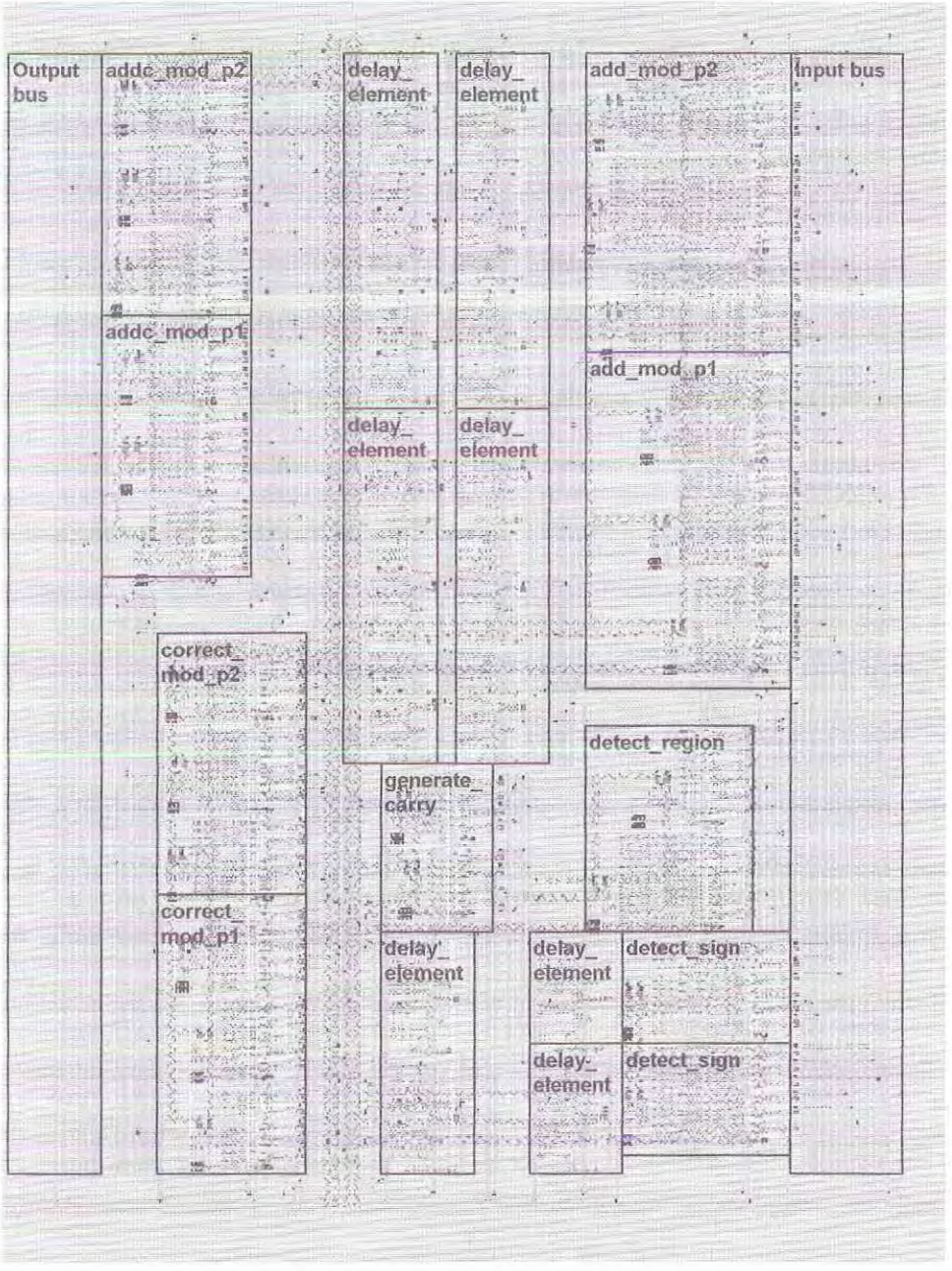

Figure 47: The radix 32 SDNR/RNS digit adder.

The digit adder was refined, so that a logic gate had a maximum delay of 20ns. Some transistors were resized so that this delay was met. The more complicated logic gates, for example, the

detect_sign component, require up to six transistors in series. In general, it was found that the more transistors in series, the greater the delay. Table 25 shows the sizing of the transistors used for the different number of the devices in series.

Table 25: Transistor sizings and propagation delays.

Number of transistors in series	Width/length transistor ratio	Delay (ns)
1	3/2	-
2	3/2	7.4
3	3/2	13.0
4	6/2	10.0
5	6/2	13.9
6	6/2	18.2

The clocking frequency chosen for the digit adder was 10MHz, operating in a four phase cycle. The digit adder implemented used types 1, 2, 3, and 4 logic gates (Figure 14). The chosen clock frequency allowed for a 25ns delay for each of the five stages in the adder. Two clock cycles were required to obtain a sum from the input operands. However, as the first set of operands are being processed in the fifth stage of the adder, a second set of operands could be passed through the first four stages, all in the one clock cycle. This allows for a degree of pipelinability.

Improvements could be made upon this scheme. The main problem is the issue of clock cycle utilisation. The required second cycle is, in effect, wasted, because it is only needed for the evaluation of the fifth stage in the digit adder. Only 25% of the second cycle is required to compute the last stage. Solutions to this problem would be to reduce the number of stages in the digit adder to four, or employ a different clocking scheme. By maintaining the four phase clocking strategy and reducing the number stages in the digit adder to four, the sum of the operands could be calculated in one cycle. For this kind of solution, the disjoint form of the digit adder could be investigated, as it has a structure (Figure 17) which is easily dissected into four stages.

The second solution, which involves changing the clocking strategy used, could result in improving performance by 100%. By using a two phase logic scheme, the clock frequency could be increased to 20MHz. For the case of the nondisjoint digit adder, three clock cycles would be required to evaluate a summation. The final cycle would be required to execute the fifth stage. This would result in a cycle utilisation of 50% for the third cycle. This is a better figure than the second cycle utilisation for the case of the four phase clocking scheme. A disjoint digit adder may prove to be a better solution in this situation, as only two clock cycles would be required to evaluate the sum of the operands. However, an analysis of a new SDNR/RNS configuration would be needed, as the disjoint digit adder can not handle nondisjoint digit sets.

To test the carry communication logic of the digit adder, two digit adders were connected together. This setup is shown in Figure 48. A simulation plan for the pair is presented in section 8-Simulations.

Figure 48: Two radix 32 SDNR/RNS digit adders.

8. Simulations

The simulation stage involved simulating the implemented radix 32 SDNR/RNS digit adder. A switch level simulator called Irsim (version 8.6) was used for this stage. There were two sets of simulations carried out. The first tested a stand alone digit adder, while the second made sure that carry information was transferred between digit adders successfully. Setup and simulation scripts were written for Irsim, so that both sets of simulations could be carried out in an efficient manner. The setup scripts set up all variables associated with the digit adders, and the simulation scripts ran through the respective test plans. The digit adder setup scripts are listed in section 10.2.1-Initialisation files, and the simulation scripts are presented in 10.2.2-Simulation files.

8.1 Single digit additions

The test plan for single digit additions, using the adder depicted in Figure 47, is listed in Table 26. All data in the table are in terms of SDNR representation. The test plan was devised in such a way that all carry in and carry out possibilities are covered. In addition, the threshold condition (t) is verified.

Table 26: Single digit addition test plan.

operand1	operand2	carry_in	sum	carry_out
$\overline{11}$	$\overline{6}$	$\overline{1}$	14	$\overline{1}$
$\overline{4}$	8	$\overline{1}$	3	0
9	10	$\overline{1}$	$\overline{14}$	1
$\overline{10}$	$\overline{17}$	0	5	$\overline{1}$
4	11	0	15	0
12	17	0	$\overline{3}$	1
$\overline{8}$	$\overline{15}$	1	10	$\overline{1}$
5	0	1	6	0
7	14	1	$\overline{10}$	1
10	6	0	16	0
7	10	0	$\overline{15}$	1
$\overline{6}$	$\overline{10}$	0	$\overline{16}$	0
$\overline{10}$	$\overline{7}$	0	15	$\overline{1}$
$\overline{17}$	$\overline{17}$	0	$\overline{2}$	$\overline{1}$
17	17	0	2	1

Figure 49 shows the Irsim output for the test plan presented in Table 26. The first operand is represented by operand1_p1 and operand1_p2, and the second operand is identified by operand2_p1 and operand2_p2. The output variables are named sum_p1 and sum_p2.

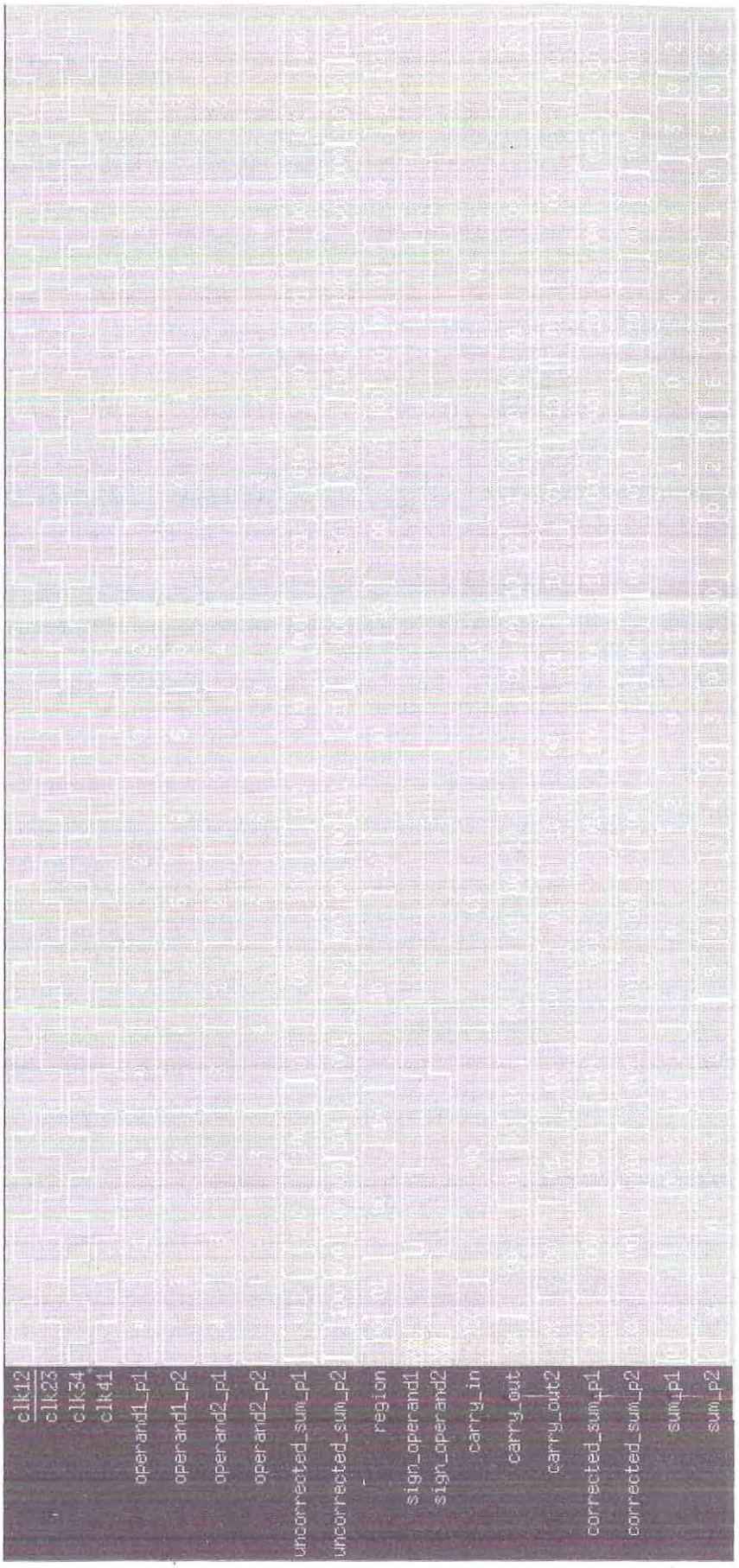

Figure 49: Irsim output for single digit additions.

8.2 Multiple digit additions

The multiple digit addition test plan was created to monitor the behaviour of two digit adders connected together (Figure 48). The test plan is shown in Table 27.

Table 27: Multiple digit addition test plan.

operand1 (MSD ... LSD)	operand2 (MSD ... LSD)	sum	Comment
(3) ($\overline{12}$)	(2) ($\bar{5}$)	(4) (15)	Tested local carry propagation. Carry from least to the next significant digit adder = -1.
(4) ($\overline{15}$)	(9) (16)	(13) (1)	Tested local carry propagation. Carry from least to the next significant digit adder = 0.
(9) (13)	($\bar{2}$) (10)	(8) ($\bar{9}$)	Tested local carry propagation. Carry from least to the next significant digit adder = 1.

The Irsim output for the test plan presented in Table 27 is shown in Figure 50. The first operand is described by word0_d0_p1 and word0_d0_p2 for the LSD, and word10_d1_p1 and word0_d1_p2 for the MSD. Likewise, the second operand is represented by word1_d0_p1 and word1_d0_p2 for the LSD, and word1_d1_p1 and word1_d1_p2 for the MSD. The final sum is assigned the variables d0_sum_p1 and d0_sum_p2 for the LSD, and d1_sum_p1 and d1_sum_p2 for the MSD.

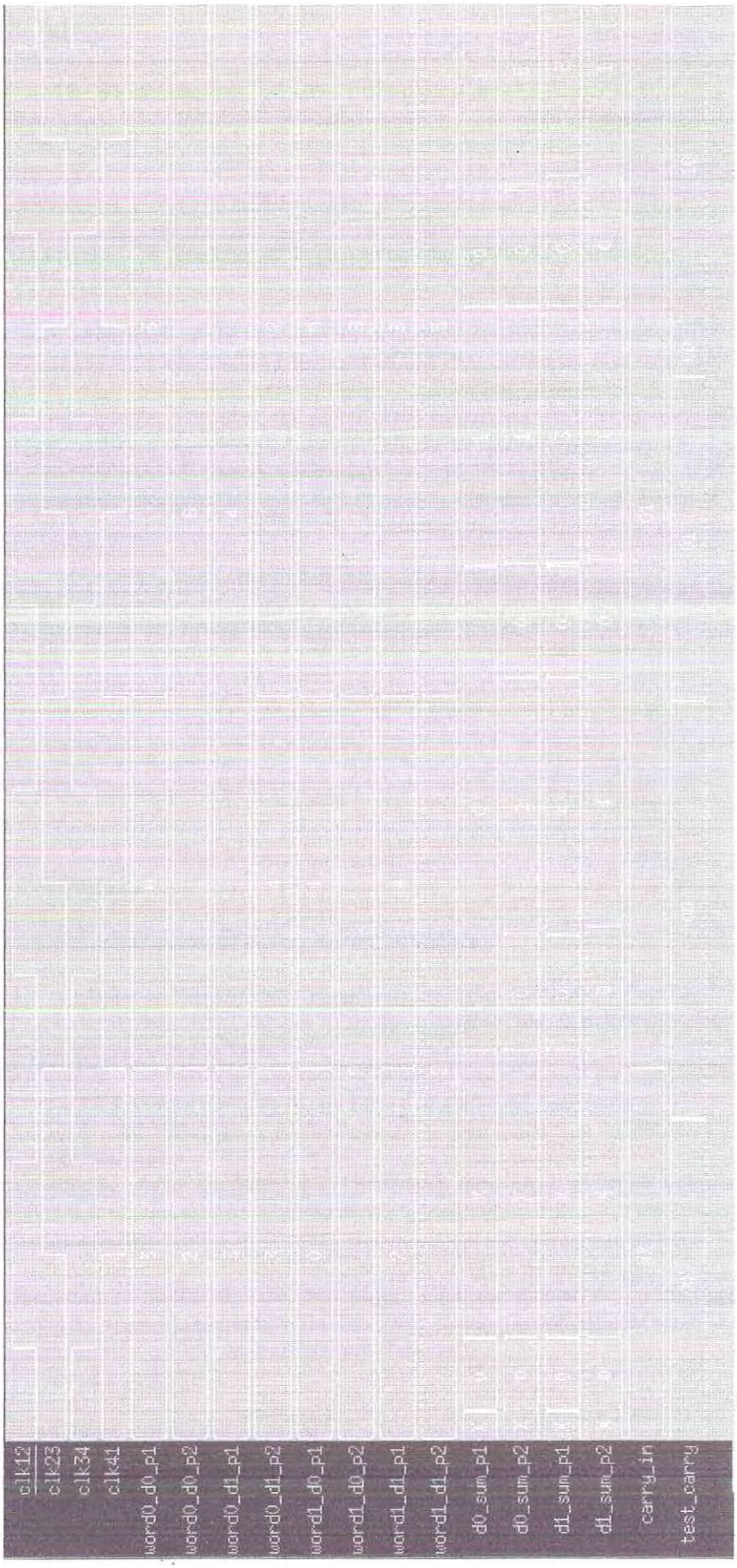

Figure 50: Irsim output for multiple digit additions.

9. Conclusions

As stated in the introduction, the aim of this project was to design and simulate a high-radix arithmetic system based on SDNR/RNS data representation. The main objectives relating to this aim were:

1. Investigating the advantages and disadvantages of high-radix SDNR/RNS arithmetic over other conventional and non-conventional schemes.
2. Determining the feasibility of implementing the SDNR/RNS arithmetic system in CMOS VLSI technology.

The main advantages of the SDNR/RNS data representation over other number systems are restricted carry propagation which allows for parallel addition and subtraction, and decomposition of complex logic networks (that is, digit sets with large a values) into modular blocks which are smaller, faster, and more manageable. However, the SDNR/RNS number system can only be applied to certain applications where additions, sign detection, and magnitude comparisons are important. For example, it was shown that the SDNR/RNS number system can not perform multiplication very efficiently. Thus, applications where multiplications are executed frequently are not very well suited to SDNR/RNS arithmetic.

In terms of VLSI feasibility, the SDNR/RNS number system shows promise. The implemented digit adder exhibited simplicity and regularity, and local communication. However, to achieve higher computation intensiveness, the SDNR/RNS addition process requires a balancing of internal processing and I/O bandwidth.

The SDNR/RNS addition algorithm could be classified as a VLSI array algorithm. By nature, the addition algorithm was parallel and pipelinable. Both of these properties are indications of a good VLSI array algorithm. To further improve upon adder performance and complexity, guidelines relating to the radix, digit set, and moduli, were presented in section 4.1-SDNR/RNS configuration analysis.

9.1 Project contribution

The original contributions of this project are as follows:

1. An analysis of SDNR/RNS parameters relating to radix, digit set, and moduli. Recommendations from various sources, regarding the SDNR/RNS parameters, were presented in a cohesive form.
2. An analysis of SDNR/RNS addition. A template was presented for verifying the design of a SDNR/RNS digit adder using the set theory of arithmetic decomposition.
3. An analysis of SDNR/RNS multiplication. It was found that SDNR/RNS multiplication was not practical.
4. An analysis of the suitability of a SDNR/RNS digit adder to VLSI technology. It was found that the characteristics of the digit adder met each of Kung's (1988) criteria.
5. Design of a SDNR/RNS digit adder, sign detection, and conversion circuits.
6. Implementation of the SDNR/RNS digit adder using VLSI technology.
7. Simulation of the SDNR/RNS digit adder. Once the simulations on the digit adder were complete, areas of deficiency were identified. Recommendations concerned with improving the performance of the digit adder were presented.

9.2 Recommendations and future research

Several issues relating to this project require further investigation. First, the performance of the implemented SDNR/RNS nondisjoint digit adder could be improved by changing the clocking strategy. It would be interesting to implement the equivalent disjoint digit adder (with the same radix and moduli set) in CMOS, and compare performance between it and the nondisjoint digit adder. An

extension of this theme would be to implement an equivalent SDNR digit adder (with the same radix and digit set), and then compare the performance against it's SDNR/RNS counterpart.

Second, the SDNR/RNS sign detector, magnitude comparator, and number system conversion circuits could be implemented in CMOS. This would allow for an analysis of the performance of these circuits.

Finally, the digit adder, and the other SDNR/RNS circuits, could be implemented in GaAs technology. This would allow for high performance ratios. In addition, the area of asynchronous logic could be explored in an attempt to eliminate the need for a clock. Asynchronous logic, or self-timed systems, do not suffer from the adverse affects of clock related problems, such as clock skew.

10. Appendices

10.1 Appendix A: Software simulation system

```
/*****************************************************************************

   Filename       :  sys_r32.h.
   Program name   :  SDNR/RNS adder (radix 32; moduli set 5, 7).
   Author         :  Paul Whyte
   Student number :  0930227
   Date           :  15/09/96
   Compiler       :  Microsoft Visual C++.

   Description

     The set of files that are included in this software simulation system
     emulate a SDNR/RNS arithmetic system.  Specifically, this system
     emulates:

       - A SDNR/RNS adder.
       - A SDNR/RNS sign detector.
       - A conventional unsigned binary to SDNR/RNS representation
         converter.
       - A SDNR/RNS representation to conventional unsigned binary
         converter.

     The system is optimised for manipulating unsigned 64-bit integers. The
     configuration for the system is as follows:

       radix          = 32
       p1             = 5
       p2             = 7
       dynamic_range  = 35
       a              = 17
       number_digits  = 13
       number_bits    = 78
       digit_set_case = 4

     The files required by this software simulation system are as follows:

       Header files:

         add_r32.h  :  Header file for add_r32.c.
         con_ctor.h :  Header file for con_ctor.c.
         con_gen.h  :  Header file for con_gen.c.
         con_rtoc.h :  Header file for con_rtoc.c.
         ds_r32.h   :  Header file for ds_r32.c.
         init_var.h :  Header file for init_var.c.
         sys_r32.h  :  (This file).  This file contains all constants and
                       data structure definitions required by the
                       software simulation system.
         user_io.h  :  Header file for user_io.c.

       Code files:

         add_r32.c  :  This file contains the functions required by the
                       SDNR/RNS adder.
         con_ctor.c :  This file contains the functions needed to convert
                       from conventional unsigned binary to SDNR/RNS
                       (redundant) representation.
         con_gen.c  :  This file contains the functions needed to convert
                       from unsigned radix 32 to unsigned binary, and vice
                       versa.
         con_rtoc.c :  This file contains the functions needed to convert
                       from SDNR/RNS (redundant) representation to
                       conventional unsigned binary.
         ds_r32.c   :  This file contains the functions required by the
                       SDNR/RNS sign detector.
         init_var.c :  This file contains functions which initialise some
                       of the data structures.
         sysa_r32.c :  This file is the SDNR/RNS adder program file.
         sysd_r32.c :  This file is the SDNR/RNS sign detector program
                       file.
         user_io.c  :  This file contains the functions needed for user
                       input and output, from and to the screen.

     Notes:
```

```
       - The characters 'dec' are used throughout this system in reference
         to a radix 32 digit.

*****************************************************************************/

#include <stdio.h>

/*****************************************************************************
  Type definitions.
*****************************************************************************/

#ifndef SYS_R32_DECLARATIONS
#define SYS_R32_DECLARATIONS

#define FALSE                   0
#define TRUE                    1
#define XDIGIT_LENGTH           16
#define DIGIT_LENGTH            XDIGIT_LENGTH - 2
#define END_OF_NUMBER           99

typedef struct
{
  unsigned b2 : 1;
  unsigned b1 : 1;
  unsigned b0 : 1;
} rns_digit_type;

typedef struct
{
  rns_digit_type p1;
  rns_digit_type p2;
} sdnr_rns_digit_type;

typedef struct
{
  unsigned b1 : 1;
  unsigned b0 : 1;
} two_bit_word_type;

typedef struct
{
  unsigned b3 : 1;
  unsigned b2 : 1;
  unsigned b1 : 1;
  unsigned b0 : 1;
} four_bit_word_type;

typedef struct
{
  unsigned b4 : 1;
  unsigned b3 : 1;
  unsigned b2 : 1;
  unsigned b1 : 1;
  unsigned b0 : 1;
} five_bit_word_type;

typedef struct
{
  unsigned b12 : 1;
  unsigned b11 : 1;
  unsigned b10 : 1;
  unsigned b9  : 1;
  unsigned b8  : 1;
  unsigned b7  : 1;
  unsigned b6  : 1;
  unsigned b5  : 1;
  unsigned b4  : 1;
  unsigned b3  : 1;
  unsigned b2  : 1;
  unsigned b1  : 1;
  unsigned b0  : 1;
} thirteen_bit_word_type;

typedef unsigned operand_conv_type [XDIGIT_LENGTH];

typedef unsigned thirteen_bit_word_array_type [XDIGIT_LENGTH];

#endif
```

```
/*****************************************************************************

  Filename:  add_r32.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"

void add_mod_p1 (rns_digit_type operand1_mod_p1, rns_digit_type operand2_mod_p1,
rns_digit_type *intermediate_sum);

void add_mod_p2 (rns_digit_type operand1_mod_p2, rns_digit_type operand2_mod_p2,
rns_digit_type *intermediate_sum);

void addc_mod_p1 (two_bit_word_type prev_carry, rns_digit_type
corrected_intermediate_sum, rns_digit_type *final_sum);

void addc_mod_p2 (two_bit_word_type prev_carry, rns_digit_type
corrected_intermediate_sum, rns_digit_type *final_sum);

void correct_mod_p1 (two_bit_word_type carry, rns_digit_type intermediate_sum,
rns_digit_type *corrected_intermediate_sum);

void correct_mod_p2 (two_bit_word_type carry, rns_digit_type intermediate_sum,
rns_digit_type *corrected_intermediate_sum);

void detect_sign (sdnr_rns_digit_type operand, unsigned *operand_sign);

void detect_region (sdnr_rns_digit_type intermediate_sum, two_bit_word_type *region);

void generate_carry (unsigned operand1_sign, unsigned operand2_sign, two_bit_word_type
region, two_bit_word_type *carry);

void add_sdnr_rns_digit (sdnr_rns_digit_type operand1_sdnr_rns,
  sdnr_rns_digit_type operand2_sdnr_rns,
  two_bit_word_type carry_in,
  sdnr_rns_digit_type *sum_sdnr_rns,
  two_bit_word_type *carry_out);
```

```
/*****************************************************************************

  Filename:  add_r32.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "add_r32.h"

void add_mod_p1 (rns_digit_type operand1_mod_p1, rns_digit_type operand2_mod_p1,
rns_digit_type *intermediate_sum)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = operand1_mod_p1.b0;
  e = operand1_mod_p1.b1;
  f = operand1_mod_p1.b2;
  a = operand2_mod_p1.b0;
  b = operand2_mod_p1.b1;
  c = operand2_mod_p1.b2;
  intermediate_sum->b2 =
    c & ~d & ~e & ~f
  | a & b & d & ~e
  | a & ~b & d & e
  | ~a & b & ~d & c
  | ~a & ~b & ~c & f;
  intermediate_sum->b1 =
    a & ~b & d & ~e
  | b & ~d & ~e & ~f
  | ~a & b & ~e & ~f
  | ~b & ~c & ~d & e
  | ~a & ~b & d & e
  | c & f
  | a & b & f;
  intermediate_sum->b0 =
    ~a & ~c & d & ~e
  | a & ~d & ~e & ~f
  | ~a & ~b & ~c & d & e
  | c & ~d & e
  | a & ~b & ~d & e
  | a & b & d & e
  | c & f
  | ~a & b & f;
}

void add_mod_p2 (rns_digit_type operand1_mod_p2, rns_digit_type operand2_mod_p2,
rns_digit_type *intermediate_sum)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = operand1_mod_p2.b0;
  e = operand1_mod_p2.b1;
  f = operand1_mod_p2.b2;
  a = operand2_mod_p2.b0;
  b = operand2_mod_p2.b1;
  c = operand2_mod_p2.b2;
  intermediate_sum->b2 =
    ~b & c & ~e & ~f
  | c & ~d & ~e & ~f
  | a & b & d & ~f
  | ~a & ~b & c & ~d & ~f
  | a & ~b & ~c & d & e
  | b & ~c & e & ~f
  | ~a & ~b & ~c & ~d & f
  | a & c & e & f
  | b & c & e & f
  | ~a & b & c & d & f
  | ~b & ~c & ~e & f
  | ~a & ~c & ~d & ~e & f;
  intermediate_sum->b1 =
```

```
    a & ~b & d & ~e & ~f
  | b & ~d & ~e & ~f
  | ~a & b & ~c & ~e & ~f
  | ~a & ~b & ~d & e
  | ~a & ~b & ~c & e
  | ~b & ~c & ~d & e & ~f
  | a & b & d & e
  | b & c & d & e
  | ~a & ~b & e & f
  | ~b & c & d & f
  | a & ~b & d & f
  | a & b & e & f
  | a & c & ~e & f
  | ~a & b & ~d & ~e & f;
  intermediate_sum->b0 =
    ~a & ~c & d & ~f
  | ~a & ~b & d & ~e & ~f
  | a & ~d & ~e & ~f
  | a & ~b & c & d & e
  | a & ~c & ~d & e & ~f
  | b & c & ~d & e
  | ~a & c & ~d & f
  | ~a & ~b & ~c & d & f
  | a & ~b & ~c & ~d & ~e & f
  | a & c & d & f
  | a & b & d & f
  | ~a & b & e & f;
}

void addc_mod_p1 (two_bit_word_type prev_carry, rns_digit_type
corrected_intermediate_sum, rns_digit_type *final_sum)

{
  unsigned a,
    b,
    c,
    d,
    e;

  d = prev_carry.b0;
  e = prev_carry.b1;
  a = corrected_intermediate_sum.b0;
  b = corrected_intermediate_sum.b1;
  c = corrected_intermediate_sum.b2;
  final_sum->b2 =
    ~a & ~b & ~c & ~d & ~e
  | c & d
  | a & b & e;
  final_sum->b1 =
    c & ~d & ~e
  | a & b & ~e
  | b & d
  | a & ~b & e
  | ~a & b & e;
  final_sum->b0 =
    c & ~d & ~e
  | a & d
  | ~a & b & ~d
  | ~a & ~c & ~d & e
  | ~a & b & e;
}

void addc_mod_p2 (two_bit_word_type prev_carry, rns_digit_type
corrected_intermediate_sum, rns_digit_type *final_sum)

{
  unsigned a,
    b,
    c,
    d,
    e;

  d = prev_carry.b0;
  e = prev_carry.b1;
  a = corrected_intermediate_sum.b0;
  b = corrected_intermediate_sum.b1;
  c = corrected_intermediate_sum.b2;
  final_sum->b2 =
    ~a & ~b & ~c & ~d & ~e
  | c & d
  | a & c
  | b & c & ~e
  | ~b & c & e
```

```
  | a & b & e;
  final_sum->b1 =
    ~a & ~b & ~d & ~e
  | a & b & ~e
  | b & d
  | a & ~b & e
  | ~a & b & ~c & ~d & e;
  final_sum->b0 =
    ~a & c & ~d & ~e
  | a & d
  | ~a & b & ~d & ~e
  | ~a & ~c & ~d & e
  | ~a & ~b & e;
}

void correct_mod_p1 (two_bit_word_type carry, rns_digit_type intermediate_sum,
rns_digit_type *corrected_intermediate_sum)

{
  unsigned a,
    b,
    c,
    d,
    e;

  d = carry.b0;
  e = carry.b1;
  a = intermediate_sum.b0;
  b = intermediate_sum.b1;
  c = intermediate_sum.b2;
  corrected_intermediate_sum->b2 =
    c & d
  | ~a & b & ~d & ~e
  | a & ~b & e;
  corrected_intermediate_sum->b1 =
    ~b & ~c & ~d & ~e
  | b & d
  | ~a & ~b & e;
  corrected_intermediate_sum->b0 =
    c & ~d & ~e
  | a & ~b & ~e
  | a & d
  | ~a & ~b & ~c & ~d & e
  | a & b & e;
}

void correct_mod_p2 (two_bit_word_type carry, rns_digit_type intermediate_sum,
rns_digit_type *corrected_intermediate_sum)

{
  unsigned a,
    b,
    c,
    d,
    e;

  d = carry.b0;
  e = carry.b1;
  a = intermediate_sum.b0;
  b = intermediate_sum.b1;
  c = intermediate_sum.b2;
  corrected_intermediate_sum->b2 =
    ~b & ~c & ~d & ~e
  | ~a & ~c & ~d & ~e
  | c & d
  | a & ~c & ~d & e
  | b & ~c & ~d & e
  | a & b & e;
  corrected_intermediate_sum->b1 =
    a & c & ~d & ~e
  | b & d & ~e
  | ~a & b & ~e
  | ~a & ~b & ~c & ~d & e
  | a & b & e
  | b & c & e;
  corrected_intermediate_sum->b0 =
    ~a & c & ~d & ~e
  | a & ~b & ~c & ~e
  | a & d
  | b & c & ~d & ~e
  | ~a & ~c & ~d & e
  | a & c & e;
}
```

```
void detect_sign (sdnr_rns_digit_type operand, unsigned *operand_sign)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = operand.p1.b0;
  e = operand.p1.b1;
  f = operand.p1.b2;
  a = operand.p2.b0;
  b = operand.p2.b1;
  c = operand.p2.b2;
  *operand_sign =
    ~a & c & ~d & ~f
  | ~a & ~b & ~c & d & ~e & ~f
  | a & c & d & ~f
  | a & b & d & ~e & ~f
  | ~a & b & ~d & ~e & ~f
  | ~a & ~b & c & e
  | ~a & ~c & d & e
  | a & ~b & ~c & ~d & e
  | a & f
  | b & c & f;

}

void detect_region (sdnr_rns_digit_type intermediate_sum, two_bit_word_type *region)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = intermediate_sum.p1.b0;
  e = intermediate_sum.p1.b1;
  f = intermediate_sum.p1.b2;
  a = intermediate_sum.p2.b0;
  b = intermediate_sum.p2.b1;
  c = intermediate_sum.p2.b2;
  region->b1 =
    ~a & ~b & ~c & d
  | ~a & c & ~d & ~f
  | a & c & d
  | a & b & d & ~e
  | ~a & b & ~d & ~e & ~f
  | a & ~b & ~c & ~d & e
  | ~a & ~c & d & e
  | a & f
  | b & c & f;
  region->b0 =
    ~a & ~b & c & d & e
  | a & b & ~d & e;
}

void generate_carry (unsigned operand1_sign, unsigned operand2_sign, two_bit_word_type
region, two_bit_word_type *carry)

{
  unsigned a,
    b,
    c,
    d;

  c = operand2_sign;
  d = operand1_sign;
  a = region.b0;
  b = region.b1;

  carry->b1 = ~c & ~d & a
  | ~c & ~d & b;
  carry->b0 = ~d & ~a & ~b
  | ~c & d & ~a
  | c & b;
}
```

```
void add_sdnr_rns_digit (sdnr_rns_digit_type operand1_sdnr_rns,
  sdnr_rns_digit_type operand2_sdnr_rns,
  two_bit_word_type carry_in,
  sdnr_rns_digit_type *sum_sdnr_rns,
  two_bit_word_type *carry_out)

{
  unsigned operand1_sign,
    operand2_sign;
  sdnr_rns_digit_type uncorrected_sdnr_rns,
    corrected_sdnr_rns;
  two_bit_word_type region;

  detect_sign (operand1_sdnr_rns, &operand1_sign);
  detect_sign (operand2_sdnr_rns, &operand2_sign);
  add_mod_p1 (operand1_sdnr_rns.p1, operand2_sdnr_rns.p1, &(uncorrected_sdnr_rns.p1));
  add_mod_p2 (operand1_sdnr_rns.p2, operand2_sdnr_rns.p2, &(uncorrected_sdnr_rns.p2));
  detect_region (uncorrected_sdnr_rns, &region);
  generate_carry (operand1_sign, operand2_sign, region, carry_out);
  correct_mod_p1 (*carry_out, uncorrected_sdnr_rns.p1, &(corrected_sdnr_rns.p1));
  correct_mod_p2 (*carry_out, uncorrected_sdnr_rns.p2, &(corrected_sdnr_rns.p2));
  addc_mod_p1 (carry_in, corrected_sdnr_rns.p1, &(sum_sdnr_rns->p1));
  addc_mod_p2 (carry_in, corrected_sdnr_rns.p2, &(sum_sdnr_rns->p2));
}
```

```
/*****************************************************************************

  Filename:  con_ctor.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"
#include "add_r32.h"

void convert_to_sdnr_rns (five_bit_word_type operand_conv_bin,
  two_bit_word_type carry_in,
  sdnr_rns_digit_type *operand_sdnr_rns,
  two_bit_word_type *carry_out);
```

```
/*****************************************************************************

  Filename:  con_ctor.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "con_ctor.h"

void convert_generate_carry (five_bit_word_type conventional_segment,
two_bit_word_type *carry)

{
  unsigned a,
    b,
    c,
    d,
    e;

  e = conventional_segment.b4;
  d = conventional_segment.b3;
  c = conventional_segment.b2;
  b = conventional_segment.b1;
  a = conventional_segment.b0;

  carry->b1 = c & e
  | d & e
  | a & e
  | b & e;
  carry->b0 = ~e
  | ~a & ~b & ~c & ~d & e;
}

void convert_mod_p1 (five_bit_word_type conventional_segment, rns_digit_type *convert)

{
  unsigned a,
    b,
    c,
    d,
    e;

  e = conventional_segment.b4;
  d = conventional_segment.b3;
  c = conventional_segment.b2;
  b = conventional_segment.b1;
  a = conventional_segment.b0;

  convert->b2 =
    ~a & ~b & c & ~d & ~e
  | a & ~b & ~c & d & ~e
  | ~a & b & c & d & ~e
  | ~a & ~b & ~c & d & e
  | a & ~b & c & d & e
  | a & b & ~c & ~d & e;
  convert->b1 =
    ~a & ~b & d & ~e
  | ~a & ~b & c & d
  | ~b & c & d & ~e
  | a & b & ~d & ~e
  | b & ~c & ~d & ~e
  | a & ~b & ~c & ~d & e
  | b & c & ~d & e
  | a & b & ~c & d & e
  | ~a & b & ~d & e;
  convert->b0 =
    ~a & ~b & ~c & d & ~e
  | a & ~c & ~d & ~e
  | a & ~b & c & d & ~e
  | a & b & ~c & ~e
  | ~a & b & c & ~d & ~e
  | ~a & ~c & ~d & e
  | ~a & ~b & c & d & e
  | a & c & ~d & e
  | a & b & c & e
  | ~a & b & ~c & e;
}

void convert_mod_p2 (five_bit_word_type conventional_segment, rns_digit_type *convert)

{
  unsigned a,
```

```
    b,
    c,
    d,
    e;

  e = conventional_segment.b4;
  d = conventional_segment.b3;
  c = conventional_segment.b2;
  b = conventional_segment.b1;
  a = conventional_segment.b0;

  convert->b2 =
    ~b & c & ~e
  | ~a & c & ~d & ~e
  | ~a & ~b & c & ~d
  | a & b & ~c & d
  | a & ~c & d & e
  | b & ~c & e;
  convert->b1 =
    a & ~b & d & ~e
  | b & ~c & ~d & ~e
  | ~a & b & ~d & ~e
  | ~a & b & ~c & ~e
  | ~a & ~b & ~d & e
  | ~a & ~b & ~c & e
  | ~b & ~c & ~d & e
  | a & b & c & e
  | a & b & d & e
  | b & c & d & e;
  convert->b0 =
    ~a & ~b & d & ~e
  | ~a & ~c & d
  | a & ~b & ~d & ~e
  | a & ~c & ~d
  | a & b & c & d
  | a & c & d & e
  | ~a & b & c & ~d & e;
}

void convert_to_sdnr_rns (five_bit_word_type operand_conv_bin,
  two_bit_word_type carry_in,
  sdnr_rns_digit_type *operand_sdnr_rns,
  two_bit_word_type *carry_out)

{
  rns_digit_type uncorrected_sdnr_rns_p1,
    uncorrected_sdnr_rns_p2,
    corrected_sdnr_rns_p1,
    corrected_sdnr_rns_p2;

  convert_generate_carry (operand_conv_bin, carry_out);
  convert_mod_p1 (operand_conv_bin, &uncorrected_sdnr_rns_p1);
  convert_mod_p2 (operand_conv_bin, &uncorrected_sdnr_rns_p2);
  correct_mod_p1 (*carry_out, uncorrected_sdnr_rns_p1, &corrected_sdnr_rns_p1);
  correct_mod_p2 (*carry_out, uncorrected_sdnr_rns_p2, &corrected_sdnr_rns_p2);
  addc_mod_p1 (carry_in, corrected_sdnr_rns_p1, &(operand_sdnr_rns->p1));
  addc_mod_p2 (carry_in, corrected_sdnr_rns_p2, &(operand_sdnr_rns->p2));
}
```

```
/*****************************************************************************

  Filename:  con_gen.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"

void convert_to_binary (unsigned digit, five_bit_word_type *operand_conv_bin);

void convert_to_decimal (five_bit_word_type sum_conv_bin, unsigned *digit);
```

```
/*****************************************************************************

  Filename:  con_gen.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "con_gen.h"

void convert_to_binary (unsigned digit, five_bit_word_type *operand_conv_bin)

{
  if ((digit & 1) == 1)
  {
    operand_conv_bin->b0 = 1;
  }
  else
  {
    operand_conv_bin->b0 = 0;
  }
  if ((digit & 2) == 2)
  {
    operand_conv_bin->b1 = 1;
  }
  else
  {
    operand_conv_bin->b1 = 0;
  }
  if ((digit & 4) == 4)
  {
    operand_conv_bin->b2 = 1;
  }
  else
  {
    operand_conv_bin->b2 = 0;
  }
  if ((digit & 8) == 8)
  {
    operand_conv_bin->b3 = 1;
  }
  else
  {
    operand_conv_bin->b3 = 0;
  }
  if ((digit & 16) == 16)
  {
    operand_conv_bin->b4 = 1;
  }
  else
  {
    operand_conv_bin->b4 = 0;
  }
}

void convert_to_decimal (five_bit_word_type sum_conv_bin, unsigned *digit)

{
  *digit = 0;
  if (sum_conv_bin.b0 == 1)
  {
    *digit = *digit + 1;
  }
  if (sum_conv_bin.b1 == 1)
  {
    *digit = *digit + 2;
  }
  if (sum_conv_bin.b2 == 1)
  {
    *digit = *digit + 4;
  }
  if (sum_conv_bin.b3 == 1)
  {
    *digit = *digit + 8;
  }
  if (sum_conv_bin.b4 == 1)
  {
    *digit = *digit + 16;
  }
}
```

```
/*****************************************************************************

  Filename:  con_rtoc.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"
#include "add_r32.h"

void convert_to_conv_bin (sdnr_rns_digit_type sum_sdnr_rns,
  unsigned sum_sign_in,
  unsigned borrow_in,
  five_bit_word_type *sum_conv_bin,
  unsigned *sum_sign_out,
  unsigned *borrow_out);
```

```
/*****************************************************************************

  Filename:  con_rtoc.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "con_rtoc.h"

void borrow_subtract (unsigned operand1, unsigned operand2, unsigned borrow_in,
unsigned *result, unsigned *borrow_out)

{
  *result = operand1 ^ operand2 ^ borrow_in;
  *borrow_out =
    ~operand1 & operand2
  | ~operand1 & borrow_in
  | operand2 & borrow_in;
}

void convert_segment (sdnr_rns_digit_type sdnr_rns_digit, five_bit_word_type
*conventional_segment)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = sdnr_rns_digit.p1.b0;
  e = sdnr_rns_digit.p1.b1;
  f = sdnr_rns_digit.p1.b2;
  a = sdnr_rns_digit.p2.b0;
  b = sdnr_rns_digit.p2.b1;
  c = sdnr_rns_digit.p2.b2;

  conventional_segment->b4 =
    ~a & c & ~d & ~f
  | ~a & ~c & d
  | a & c & d
  | a & b & d & ~e
  | ~a & b & ~c & ~e & ~f
  | a & ~c & ~d & e
  | a & f
  | b & c & f;
  conventional_segment->b3 =
    ~a & ~b & c & d
  | a & ~c & ~d & ~e & ~f
  | a & b & ~e & ~f
  | b & ~c & ~d & ~e & ~f
  | c & e
  | ~b & d & e
  | ~a & b & f
  | ~b & ~c & ~d & f;
  conventional_segment->b2 =
    ~b & c & ~d & ~f
  | a & ~b & ~d & ~e & ~f
  | a & c & ~f
  | a & b & d & ~e
  | b & c & d
  | ~a & ~b & ~d & e
  | ~b & c & e
  | ~a & b & d & e
  | ~a & ~b & f
  | a & b & f
  | b & c & f;
  conventional_segment->b1 =
    ~a & ~b & c & ~e & ~f
  | ~a & ~b & d & ~e
  | ~b & c & d
  | c & d & ~e
  | a & ~c & ~d & ~e & ~f
  | b & ~c & ~d & ~e & ~f
  | ~b & ~c & ~d & e
  | ~a & ~c & ~d & e
  | a & b & d & e
  | b & c & f
  | ~b & ~c & ~d & ~e & f;
  conventional_segment->b0 =
    ~b & c & d & ~e
```

```
    | a & ~b & ~e & ~f
    | ~a & b & ~d & ~e
    | ~a & ~b & e
    | ~b & ~c & ~d & e
    | a & b & e
    | b & c & d & e
    | b & f;
}

void convert_to_conv_bin (sdnr_rns_digit_type sum_sdnr_rns,
  unsigned sum_sign_in,
  unsigned borrow_in,
  five_bit_word_type *sum_conv_bin,
  unsigned *sum_sign_out,
  unsigned *borrow_out)

{
  five_bit_word_type uncorrected_conv_bin;
  unsigned borrow;
  unsigned a,
    b,
    c,
    d,
    e;

  detect_sign (sum_sdnr_rns, sum_sign_out);
  convert_segment (sum_sdnr_rns, &uncorrected_conv_bin);
  borrow_subtract (uncorrected_conv_bin.b0, sum_sign_in, borrow_in, &a, &borrow);
  borrow_subtract (uncorrected_conv_bin.b1, 0, borrow, &b, &borrow);
  borrow_subtract (uncorrected_conv_bin.b2, 0, borrow, &c, &borrow);
  borrow_subtract (uncorrected_conv_bin.b3, 0, borrow, &d, &borrow);
  borrow_subtract (uncorrected_conv_bin.b4, 0, borrow, &e, borrow_out);
  sum_conv_bin->b0 = a;
  sum_conv_bin->b1 = b;
  sum_conv_bin->b2 = c;
  sum_conv_bin->b3 = d;
  sum_conv_bin->b4 = e;
}
```

```
/*****************************************************************************

  Filename:  ds_r32.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"

void detect_zero (sdnr_rns_digit_type operand, unsigned *operand_zero);

void mux_get_position (thirteen_bit_word_array_type word_content, four_bit_word_type
select, unsigned *position);

Void mux_select (thirteen_bit_word_array_type word_content, four_bit_word_type
*select);
```

```
/*****************************************************************************

  Filename:  ds_r32.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "ds_r32.h"

void detect_zero (sdnr_rns_digit_type operand, unsigned *operand_zero)

{
  unsigned a,
    b,
    c,
    d,
    e,
    f;

  d = operand.p1.b0;
  e = operand.p1.b1;
  f = operand.p1.b2;
  a = operand.p2.b0;
  b = operand.p2.b1;
  c = operand.p2.b2;

  *operand_zero =
    a
  | b
  | c
  | d
  | e
  | f;
}

void mux_get_position (thirteen_bit_word_array_type word_content, four_bit_word_type
select, unsigned *position)

{
  int index = 0;

  if (select.b0 == 1)
  {
    index = index + 1;
  }
  if (select.b1 == 1)
  {
    index = index + 2;
  }
  if (select.b2 == 1)
  {
    index = index + 4;
  }
  if (select.b3 == 1)
  {
    index = index + 8;
  }
  *position = word_content [index];
}

void mux_select (thirteen_bit_word_array_type word_content, four_bit_word_type
*select)

{
  unsigned d0,
  d1,
  d2,
  d3,
  d4,
  d5,
  d6,
  d7,
  d8,
  d9,
  d10,
  d11,
  d12;

  d0 = word_content [0];
  d1 = word_content [1];
  d2 = word_content [2];
  d3 = word_content [3];
```

```
  d4 = word_content [4];
  d5 = word_content [5];
  d6 = word_content [6];
  d7 = word_content [7];
  d8 = word_content [8];
  d9 = word_content [9];
  d10 = word_content [10];
  d11 = word_content [11];
  d12 = word_content [12];
  select->b3 =
  d12 | ~d12 & (d11 | ~d11 & (d10 | ~d10 & (d9 | ~d9 & d8)));
  select->b2 =
  d12 | ~d12 & (~d11 & (~d10 & (~d9 & (~d8 & (d7 | ~d8 & (~d7 & (d6 | ~d6 & (d5 | ~d5
& d4)))))));
  select->b1 =
  ~d12 & (d11 | ~d11 & (d10 | ~d10 & (~d9 & (~d8 & (d7 | ~d7 & (d6 | ~d6 & (~d5 & (~d4
& (d3 | ~d3 & d2)))))));
  select->b0 =
  ~d12 & (d11 | ~d11 & (~d10 & (d9 | ~d9 & (~d8 & (d7 | ~d7 & (~d6 & (d5 | ~d5 & (~d4
& (d3 | ~d3 & ~d2 & d1)))))));
}
```

```
/*****************************************************************************

  Filename:  init_var.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"

void init_carry (two_bit_word_type *carry);
void init_operand_conv_dec (operand_conv_type operand_conv_dec);
```

```
/*****************************************************************************

  Filename:  init_var.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "init_var.h"

void init_carry (two_bit_word_type *carry)

{
  carry->b1 = 0;
  carry->b0 = 1;
}

void init_operand_conv_dec (operand_conv_type operand_conv_dec)

{
  int index;

  for (index = 0; index <= DIGIT_LENGTH; index++)
  {
    operand_conv_dec [index] = 0;
  }
}
```

```
/*****************************************************************************

  Filename:  sysa_r32.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"
#include "init_var.h"
#include "user_io.h"
#include "add_r32.h"
#include "con_ctor.h"
#include "con_rtoc.h"
#include "con_gen.h"

void main ()

{
  operand_conv_type operand1_conv_dec,
    operand2_conv_dec,
    sum_conv_dec;
  five_bit_word_type operand1_conv_bin,
    operand2_conv_bin,
    sum_conv_bin;
  sdnr_rns_digit_type operand1_sdnr_rns,
    operand2_sdnr_rns,
    sum_sdnr_rns;
  int index = 0,
    exit_loop = FALSE;
  two_bit_word_type operand1_carry_in,
    operand1_carry_out,
    operand2_carry_in,
    operand2_carry_out,
    sum_carry_in,
    sum_carry_out;
  unsigned sum_sign_in,
    sum_borrow_in,
    sum_sign_out,
    sum_borrow_out;

  while (exit_loop == FALSE)
  {
    init_operand_conv_dec (operand1_conv_dec);
    init_operand_conv_dec (operand2_conv_dec);
    prompt_operand (1, operand1_conv_dec);
    prompt_operand (2, operand2_conv_dec);
    init_carry (&operand1_carry_in);
    init_carry (&operand2_carry_in);
    init_carry (&sum_carry_in);
    sum_sign_in = 0;
    sum_borrow_in = 0;
    for (index = 0; index <= DIGIT_LENGTH; index++)
    {
      convert_to_binary (operand1_conv_dec [index], &operand1_conv_bin);
      convert_to_binary (operand2_conv_dec [index], &operand2_conv_bin);

      convert_to_sdnr_rns (operand1_conv_bin, operand1_carry_in, &operand1_sdnr_rns,
&operand1_carry_out);
      operand1_carry_in = operand1_carry_out;
      convert_to_sdnr_rns (operand2_conv_bin, operand2_carry_in, &operand2_sdnr_rns,
&operand2_carry_out);
      operand2_carry_in = operand2_carry_out;

      add_sdnr_rns_digit (operand1_sdnr_rns, operand2_sdnr_rns, sum_carry_in,
&sum_sdnr_rns, &sum_carry_out);
      sum_carry_in = sum_carry_out;

      convert_to_conv_bin (sum_sdnr_rns, sum_sign_in, sum_borrow_in, &sum_conv_bin,
&sum_sign_out, &sum_borrow_out);
      sum_sign_in = sum_sign_out;
      sum_borrow_in = sum_borrow_out;

      convert_to_decimal (sum_conv_bin, &(sum_conv_dec [index]));
    }
    display_conv_dec_segment (sum_conv_dec);
    exit_loop = prompt_repeat ();
  }
}
```

```
/*****************************************************************************

  Filename:  sysd_r32.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"
#include "init_var.h"
#include "user_io.h"
#include "add_r32.h"
#include "ds_r32.h"
#include "con_ctor.h"
#include "con_rtoc.h"
#include "con_gen.h"

void main ()

{
  operand_conv_type operand_conv_dec;
  thirteen_bit_word_array_type sign_signature,
    zero_signature;
  five_bit_word_type operand_conv_bin;
  four_bit_word_type select;
  two_bit_word_type operand_carry_in,
    operand_carry_out;
  sdnr_rns_digit_type operand_sdnr_rns;
  unsigned operand_sign,
    operand_zero,
    sign,
    zero;
  int index = 0,
    exit_loop = FALSE;

  while (exit_loop == FALSE)
  {
    init_operand_conv_dec (operand_conv_dec);
    prompt_operand (1, operand_conv_dec);
    init_carry (&operand_carry_in);
    for (index = 0; index <= DIGIT_LENGTH; index++)
    {
      convert_to_binary (operand_conv_dec [index], &operand_conv_bin);
      convert_to_sdnr_rns (operand_conv_bin, operand_carry_in, &operand_sdnr_rns,
&operand_carry_out);
      operand_carry_in = operand_carry_out;
      detect_sign (operand_sdnr_rns, &operand_sign);
      detect_zero (operand_sdnr_rns, &operand_zero);
      sign_signature [index] = operand_sign;
      zero_signature [index] = operand_zero;
    }
    mux_select (zero_signature, &select);
    mux_get_position (sign_signature, select, &sign);
    mux_get_position (zero_signature, select, &zero);
    display_sign (sign, zero);
    exit_loop = prompt_repeat ();
  }
}
```

```
/*****************************************************************************

  Filename:  user_io.h.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include <stdio.h>
#include "sys_r32.h"

void display_conv_bin_segment (int digit_index,
  five_bit_word_type conv_bin_segment);

void display_conv_dec_segment (operand_conv_type conv_dec_segment);

void display_sdnr_rns_digit (int digit_index,
  sdnr_rns_digit_type sdnr_rns_digit);

void display_sign (unsigned sign, unsigned zero);

void prompt_operand (int operand_index, operand_conv_type operand_conv_dec);

int prompt_repeat ();
```

```
/*****************************************************************************

  Filename:  user_io.c.

  Refer to sys_r32.h for documentation.

*****************************************************************************/

#include "user_io.h"

void display_conv_bin_segment (int digit_index,
  five_bit_word_type conv_bin_segment)

{
  printf ("Conventional binary segment %d :  %d%d%d%d%d\n", digit_index,
    conv_bin_segment.b4,
    conv_bin_segment.b3,
    conv_bin_segment.b2,
    conv_bin_segment.b1,
    conv_bin_segment.b0);
}

void display_conv_dec_segment (operand_conv_type conv_dec_segment)

{
  int index;

  printf ("The sum of the operands is:  ");
  for (index = DIGIT_LENGTH - 2; index >= 0; index--)
  {
    printf ("%2d ", conv_dec_segment [index]);
  }
  printf ("\n");
}

void display_sdnr_rns_digit (int digit_index,
  sdnr_rns_digit_type sdnr_rns_digit)

{
  printf ("SDNR/RNS digit %d <p1, p2>:  <%d%d%d, %d%d%d>\n", digit_index,
    sdnr_rns_digit.p1.b2,
    sdnr_rns_digit.p1.b1,
    sdnr_rns_digit.p1.b0,
    sdnr_rns_digit.p2.b2,
    sdnr_rns_digit.p2.b1,
    sdnr_rns_digit.p2.b0);
}

void display_sign (unsigned sign, unsigned zero)

{
  printf ("The sign of the operand is:  ");
  if (sign == 0)
  {
    printf ("Positive.\n");
  }
  else
  {
    printf ("Negative.\n");
  }
  printf ("The operand is            :  ");
  if (zero == 0)
  {
    printf ("Zero.\n");
  }
  else
  {
    printf ("Not zero.\n");
  }
}

void prompt_operand (int operand_index, operand_conv_type operand_conv_dec)

{
  int index,
    reverse_index,
    forward_index;
  operand_conv_type temp_operand_conv_dec;

  printf ("Enter operand %d:  ", operand_index);
  scanf ("%d", &(temp_operand_conv_dec [0]));
  for (index = 1; temp_operand_conv_dec [index - 1] != END_OF_NUMBER; index++)
  {
```

```
    scanf ("%d", &(temp_operand_conv_dec [index]));
  }
  getchar ();
  for (reverse_index = index - 2, forward_index = 0; reverse_index >= 0;
reverse_index--, forward_index++)
  {
    operand_conv_dec [forward_index] = temp_operand_conv_dec [reverse_index];
  }
}

int prompt_repeat ()

{
  int user_exit = FALSE;
  char response = '\0';

  while ((response != 'y') && (response != 'Y') && (response != 'n') && (response !=
'N'))
  {
    printf ("Do you want to exit?  ");
    response = getchar ();
    getchar ();
  }
  if ((response == 'y') || (response == 'Y'))
  {
    user_exit = TRUE;
  }
  return user_exit;
}
```

10.2 Appendix B: Irsim simulation scripts

10.2.1 Initialisation files

```
| Filename       :  iadder.cmd
| Author         :  Paul Whyte
| Student number :  0930227
| Date           :  9/10/96
|
| Description    :
|
|   This is a command file for irsim.  This file initialises all parameters
|   for a SDNR/RNS digit adder.

clock clk12 0 1 1 0
clock clk23 0 0 1 1
clock clk34 1 0 0 1
clock clk41 1 1 0 0
vector operand1_p1 operand1_p1_b2 operand1_p1_b1 operand1_p1_b0
vector operand1_p2 operand1_p2_b2 operand1_p2_b1 operand1_p2_b0
vector operand2_p1 operand2_p1_b2 operand2_p1_b1 operand2_p1_b0
vector operand2_p2 operand2_p2_b2 operand2_p2_b1 operand2_p2_b0
vector uncorrected_sum_p1 uncorrected_sum_p1_b2 uncorrected_sum_p1_b1
uncorrected_sum_p1_b0
vector uncorrected_sum_p2 uncorrected_sum_p2_b2 uncorrected_sum_p2_b1
uncorrected_sum_p2_b0
vector region region_b1 region_b0
vector carry_in carry_in_b1 carry_in_b0
vector carry_out carry_out_b1 carry_out_b0
vector carry_out2 carry_out2_b1 carry_out2_b0
vector corrected_sum_p1 corrected_sum_p1_b2 corrected_sum_p1_b1 corrected_sum_p1_b0
vector corrected_sum_p2 corrected_sum_p2_b2 corrected_sum_p2_b1 corrected_sum_p2_b0
vector sum_p1 sum_p1_b2 sum_p1_b1 sum_p1_b0
vector sum_p2 sum_p2_b2 sum_p2_b1 sum_p2_b0
w clk12 clk23 clk34 clk41
w operand1_p1
w operand1_p2
w operand2_p1
w operand2_p2
w uncorrected_sum_p1
w uncorrected_sum_p2
w region
w sign_operand1
w sign_operand2
w carry_in
w carry_out
w carry_out2
w corrected_sum_p1
w corrected_sum_p2
w sum_p1
w sum_p2
ana clk12 clk23 clk34 clk41
ana operand1_p1
ana operand1_p2
ana operand2_p1
ana operand2_p2
ana uncorrected_sum_p1
ana uncorrected_sum_p2
ana region
ana sign_operand1
ana sign_operand2
ana carry_in
ana carry_out
ana carry_out2
ana corrected_sum_p1
ana corrected_sum_p2
ana sum_p1
ana sum_p2
stepsize 25
```

```
| Filename        :  i2adder.cmd
| Author          :  Paul Whyte
| Student number  :  0930227
| Date            :  9/10/96
|
| Description     :
|
|   This is a command file for irsim.  This file initialises all parameters
|   for two SDNR/RNS digit adders.

clock clk12 0 1 1 0
clock clk23 0 0 1 1
clock clk34 1 0 0 1
clock clk41 1 1 0 0
vector word0_d0_p1 word0_d0_p1_b2 word0_d0_p1_b1 word0_d0_p1_b0
vector word0_d0_p2 word0_d0_p2_b2 word0_d0_p2_b1 word0_d0_p2_b0
vector word0_d1_p1 word0_d1_p1_b2 word0_d1_p1_b1 word0_d1_p1_b0
vector word0_d1_p2 word0_d1_p2_b2 word0_d1_p2_b1 word0_d1_p2_b0
vector word1_d0_p1 word1_d0_p1_b2 word1_d0_p1_b1 word1_d0_p1_b0
vector word1_d0_p2 word1_d0_p2_b2 word1_d0_p2_b1 word1_d0_p2_b0
vector word1_d1_p1 word1_d1_p1_b2 word1_d1_p1_b1 word1_d1_p1_b0
vector word1_d1_p2 word1_d1_p2_b2 word1_d1_p2_b1 word1_d1_p2_b0
vector d0_sum_p1 d0_sum_p1_b2 d0_sum_p1_b1 d0_sum_p1_b0
vector d0_sum_p2 d0_sum_p2_b2 d0_sum_p2_b1 d0_sum_p2_b0
vector d1_sum_p1 d1_sum_p1_b2 d1_sum_p1_b1 d1_sum_p1_b0
vector d1_sum_p2 d1_sum_p2_b2 d1_sum_p2_b1 d1_sum_p2_b0
vector carry_in carry_in_b1 carry_in_b0
vector test_carry test_carry_b1 test_carry_b0
w clk12 clk23 clk34 clk41
w word0_d0_p1
w word0_d0_p2
w word0_d1_p1
w word0_d1_p2
w word1_d0_p1
w word1_d0_p2
w word1_d1_p1
w word1_d1_p2
w d0_sum_p1
w d0_sum_p2
w d1_sum_p1
w d1_sum_p2
w carry_in
w test_carry
ana clk12 clk23 clk34 clk41
ana word0_d0_p1
ana word0_d0_p2
ana word0_d1_p1
ana word0_d1_p2
ana word1_d0_p1
ana word1_d0_p2
ana word1_d1_p1
ana word1_d1_p2
ana d0_sum_p1
ana d0_sum_p2
ana d1_sum_p1
ana d1_sum_p2
ana carry_in
ana test_carry
stepsize 25
```

10.2.2 Simulation files

```
| Filename        :  sim_op2.cmd
| Author          :  Paul Whyte
| Student number  :  0930227
| Date            :  9/10/96
|
| Description     :
|
|   This is a command file for irsim.  This file simulates a stream of operand
|   inputs into a SDNR/RNS digit adder.
|
| Notes           :
|
|   The command file iadder.cmd must be run first before this file can be run
|   in the irsim environment.

| Set operand 1     = -11
| Set operand 2     = - 6
| Set carry in      = - 1
| Expect sum.       =  14
| Expect carry out = - 1

set operand1_p1 100
set operand1_p2 011
set operand2_p1 100
set operand2_p2 001
c
set carry_in 00

| Set operand 1     = - 4
| Set operand 2     =   8
| Set carry in      = - 1
| Expect sum        =   3
| Expect carry out =    0

set operand1_p1 001
set operand1_p2 011
set operand2_p1 011
set operand2_p2 001
c

| Set operand 1     =   9
| Set operand 2     =  10
| Set carry in      = - 1
| Expect sum        = -14
| Expect carry out =    1

set operand1_p1 100
set operand1_p2 010
set operand2_p1 000
set operand2_p2 011
c

| Set operand 1     = -10
| Set operand 2     = -17
| Set carry in      =   0
| Expect sum        =   5
| Expect carry out = - 1

set operand1_p1 000
set operand1_p2 100
set operand2_p1 011
set operand2_p2 100
c
set carry_in 01

| Set operand 1     =   4
| Set operand 2     =  11
| Set carry in      =   0
| Expect sum        =  15
| Expect carry out =    0

set operand1_p1 100
set operand1_p2 100
set operand2_p1 001
set operand2_p2 100
c

| Set operand 1     =  12
```

```
| Set operand 2     =  17
| Set carry in      =   0
| Expect sum        = - 3
| Expect carry out =   1

set operand1_p1 010
set operand1_p2 101
set operand2_p1 010
set operand2_p2 011
c

| Set operand 1     = - 8
| Set operand 2     = -15
| Set carry in      =   1
| Expect sum        = - 3
| Expect carry out = - 1

set operand1_p1 010
set operand1_p2 110
set operand2_p1 000
set operand2_p2 110
c
set carry_in 10

| Set operand 1     =   5
| Set operand 2     =   0
| Set carry in      =   1
| Expect sum        =   6
| Expect carry out =   0

set operand1_p1 000
set operand1_p2 101
set operand2_p1 000
set operand2_p2 000
c

| Set operand 1     =   7
| Set operand 2     =  14
| Set carry in      =   1
| Expect sum        = -10
| Expect carry out =   1

set operand1_p1 010
set operand1_p2 000
set operand2_p1 100
set operand2_p2 000
c

| Set operand 1     =  10
| Set operand 2     =   6
| Set carry in      =   0
| Expect sum        =  16
| Expect carry out =   0

set operand1_p1 000
set operand1_p2 011
set operand2_p1 001
set operand2_p2 110
c
set carry_in 01

| Set operand 1     =   7
| Set operand 2     =  10
| Set carry in      =   0
| Expect sum        = -15
| Expect carry out =   1

set operand1_p1 010
set operand1_p2 000
set operand2_p1 000
set operand2_p2 011
c

| Set operand 1     = - 6
| Set operand 2     = -10
| Set carry in      =   0
| Expect sum        = -16
| Expect carry out =   0

set operand1_p1 100
set operand1_p2 001
set operand2_p1 000
set operand2_p2 100
```

```
c

| Set operand 1     = -10
| Set operand 2     = - 7
| Set carry in      =   0
| Expect sum        =  15
| Expect carry out = - 1

set operand1_p1 000
set operand1_p2 100
set operand2_p1 011
set operand2_p2 000
c

| Set operand 1     = -17
| Set operand 2     = -17
| Set carry in      =   0
| Expect sum        = - 2
| Expect carry out = - 1

set operand1_p1 011
set operand1_p2 100
set operand2_p1 011
set operand2_p2 100
c

| Set operand 1     =  17
| Set operand 2     =  17
| Set carry in      =   0
| Expect sum        =   2
| Expect carry out =   1

set operand1_p1 010
set operand1_p2 011
set operand2_p1 010
set operand2_p2 011
c
c
```

```
| Filename       :  sim2_op2.cmd
| Author         :  Paul Whyte
| Student number :  0930227
| Date           :  9/10/96
|
| Description    :
|
|   This is a command file for irsim.  This file simulates a stream of operand
|   inputs into two SDNR/RNS digit adders.
|
| Notes          :
|
|   The command file i2adder.cmd must be run first before this file can be run
|   in the irsim environment.

|                      MSD   LSD
| Set operand 1    = (  3) (-12)
| Set operand 2    = (  2) (- 5)
| Set carry in     =       (  0)
| Expect sum       = (  4) ( 15)
| Expect carry out =       (  0)

set word0_d1_p1 011
set word0_d1_p2 011
set word0_d0_p1 011
set word0_d0_p2 010
set word1_d1_p1 010
set word1_d1_p2 010
set word1_d0_p1 000
set word1_d0_p2 010
c
set carry_in 01

|                      MSD   LSD
| Set operand 1    = (  4) (-15)
| Set operand 2    = (  9) ( 16)
| Set carry in     =       (  0)
| Expect sum       = ( 13) (  1)
| Expect carry out =       (  0)

set word0_d1_p1 100
set word0_d1_p2 100
set word0_d0_p1 000
set word0_d0_p2 110
set word1_d1_p1 100
set word1_d1_p2 010
set word1_d0_p1 001
set word1_d0_p2 010
c

|                      MSD   LSD
| Set operand 1    = (  9) ( 13)
| Set operand 2    = (- 2) ( 10)
| Set carry in     =       (  0)
| Expect sum       = (  8) ( -9)
| Expect carry out =       (  0)

set word0_d1_p1 100
set word0_d1_p2 010
set word0_d0_p1 011
set word0_d0_p2 110
set word1_d1_p1 011
set word1_d1_p2 101
set word1_d0_p1 000
set word1_d0_p2 011
c
c
```

11. References

Abdallah, M. & Skavantzos, A. (1995). A systematic approach for selecting practical moduli sets for residue number systems. *Proceedings of the 27th Southeastern Symposium on System Theory* (pp. 445 - 449). Starkville, MS: Mississippi State University College of Engineering.

Avizienis, A. (1961). Signed-digit number representation for fast parallel arithmetic. In E. E. Swartzlander Jr (Ed.). *Computer arithmetic: Volume II* (pp. 54 - 65). Los Alamitos, CA: IEEE Computer Society Press.

Carter, T. M. & Robertson, J. E. (1990). The set theory of arithmetic decomposition. *IEEE Transactions on Computers*, 39 (8), 993 - 1005.

Ercegovac, M. D. & Lang, T. (1987). On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, 36 (7), 895 - 897.

Glasser, L. A., & Dobberpuhl, D. W. (1988). *The design and analysis of VLSI circuits*. Reading, MA: Addison-Wesley Publishing Company.

Hayes, J. P. (1993). *Introduction to digital logic design*. Reading, MA: Addison-Wesley Publishing Company.

Hwang, K. & Briggs, F. A. (1984). *Computer architecture and parallel processing*. New York, NY: McGraw-Hill Publishing Company.

Hwang, K. (Ed.) & Degroot, D. (Ed.). (1989). *Parallel processing for supercomputers & artificial intelligence*. New York, NY: McGraw-Hill Publishing Company.

Kuczborski, W. (1993). *Real-time morphological processors based on redundant number representation*. Unpublished doctoral dissertation, University of Western Australia, Perth, Western Australia.

Kuczborski, W., Attikiouzel, Y. & Crebbin, G. (1994). Decomposition of logic networks with emphasis on signed digit arithmetic systems. *IEE Proceedings, Circuits Devices and Systems*, 141 (4), 307 - 314.

Kung, H. T. (1982). Why systolic architectures? *IEEE Computer, 15* (1), 37 - 46.

Kung, S. Y. (1988). *VLSI array processors*. Englewood Cliffs, NJ: Prentice Hall.

Luba, T. (1994). Multi-level logic synthesis based on decomposition. *Microprocessors and Microsystems*, 18 (8), 429 - 437.

Pedler, P. (1993). *Mathematics for computing and engineering*. Perth, WA: Edith Cowan University.

Pucknell, D. A. & Eshraghian, K. (1994). *Basic VLSI design* (3rd ed.). Sydney, NSW: Prentice Hall.

Rajashekhara, T. N. & Nale, A. S. (1990). Conversion from signed-digit to radix complement representation. *International Journal of Electronics*, 60 (6), 717 - 721.

Ramamoorthy, P. A., Potu, B., & Govind, G. (1988). DSP system architecture using signed-digit number representation. *ICASSP 88: 1988 International Conference on Acoustics, Speech, and Signal Processing* (pp. 1702 - 1705). New York, NY: IEEE.

Spaniol, O. (1981). *Computer arithmetic: Logic and design*. Chichester: John Wiley & Sons.

Streetman, B. G. (1990). *Solid state electronic devices* (3rd ed.). Englewood Cliffs, NJ: Prentice Hall.

Taylor, F. J. (1984). Residue arithmetic: A tutorial with examples. *IEEE Computer*, 50 - 62.

Waser, S. & Flynn, M. J. (1982). *Introduction to arithmetic for digital systems designers*. Forth Worth, TX: Saunders College Publishing.

Weste, N. H. E. & Eshraghian, K. (1994). *Principles of CMOS VLSI design: A systems perspective* (2nd ed.). Reading, MA: Addison-Wesley Publishing Company.

Yang, C., Lu, H. C. & Gilbert, D. E. (1991). An investigation into the implementation costs of residue and high radix arithmetic. *Proceedings of the Twenty-First International Symposium on Multiple-Valued Logic* (pp. 364 - 371). Los Alamitos, CA: IEEE Computer Society Press.