

DATA CONVERSION IN RESIDUE NUMBER SYSTEM

Enrolment No. - 14102227

Name - Abhishek Chauhan

Enrolment No. -14102007

Name - Aman Aggarwal

Name of the Supervisor - Shruti Sabharwal



2016-2017

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

DEPARTMENT OF ELECTRONICS & COMMUNICATION

JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA

CONTENTS

Certificate	4
Acknowledgement	5
Summary	6
List of Figures	7
List of Acronyms	8
1. Chapter 1	9
1.1 General RNS Based Processor Structure	10
1.2 Conventional Number System	11
1.3 Residue Number Systems	12
1.4 Mathematical Fundamentals:	12
1.4.1 Basic Definitions and Congruences	12
1.4.2 Basic Algebraic Operations	12
1.5 Conversion	14
2. Chapter 2	15
2.1 Forward Conversion from Binary to RNS Representation	15
2.2 Parallel Conversion	15
2.3 Special Moduli-Set Forward Converter	15
2.31 The Special Moduli-Set $\{2^n - 1, 2^n, 2^n + 1\}$	16
2.4 Modulo Adder For Special Moduli	17
2.41 Modulo 2^n Adder	19
2.42 Modulo $2^n - 1$ Adder	19
2.43 Modulo $2^n + 1$ Adder	22

3. Chapter 3	28
3.1 Introduction	28
3.2 Chinese Remainder Theorem	30
3.21 Modulo-m-adder	30
3.22 Carry Select Adder Used as subblock in modulo-m-adder	32
4. Results and Conclusions	36
4.1 Results	36
4.2 Conclusions	40

CERTIFICATE

This is to certify that the work titled “**Data Conversion in Residue Number System**” submitted by “**Abhishek Chauhan**” and “**Aman Aggarwal**” in partial fulfillment for the award of degree of Bachelors of Technology of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of the Supervisor _____

Name of the Supervisor _____

Designation _____

Date _____

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompanies the successful completion of the project would be incomplete without mentioning the names of the people who have helped us achieve it.

So we take this opportunity to thank my Supervisor “Shruti Sabharwal”, she has been a mentor and a guide along the entire journey of this project. We thank her for his immense support and guidance. Without her the project could never have been a success.

We also thank Electronics & Communication Department, IIIT, Noida, for providing us the right facilities and environment to complete my project.

We also thank our friends and family members, for constantly motivating us and co-operating with us in the progress of the project.

Name of the Student Abhishek Chauhan

Date _____

Name of the Student Aman Aggarwal

Date _____

SUMMARY

This project aims at conversion of data between binary and residue number system. It involves optimising circuits at gate level to implement fast working converter. Different schemes and architectures for forward data conversion from conventional representation to RNS representation are discussed. This process is considered as a pre-processing step to encode the data into residue form with respect to some given moduli. The overhead of this conversion stage has to be minimized to exploit the advantages of the RNS.

Different schemes and architectures for reverse data conversion from RNS representation to conventional representation are discussed. After the residue encoded data is processed by the RNS processor, they have to be converted back into conventional representation. This process is considered as a post-processing step. The reverse conversion is one of the most difficult RNS operations and has been a major, if not the major, limiting factor to a wider use of RNS.

All reverse conversion algorithms are, in a way or another, based on Chinese Remainder Theorem (CRT) or Mixed-Radix Conversion (MRC). Various schemes for the implementation of these algorithms when the output is in binary representation are discussed.

In this project, we focused on the CRT algorithmic side of the design where most of the available and proposed data converters in RNS rely on similar data conversion techniques in digital systems.

LIST OF FIGURES

Fig.1.1 General Structure of RNS based processor [2]

Fig 2.1 Parallel Forward converter [3]

Fig 2.2 $\{2^n - 1, 2^n, 2^n + 1\}$ forward converter [3]

Fig 2.3 Modulo $2^n - 1$ carry look ahead adder [3]

Fig 2.4 Modulo $2^n + 1$ Adder [2]

Fig 2.5 Cell Definition [4]

Fig 2.6 Build 8 bit Sklansky Adder : Step 1 [4]

Fig 2.7 Build 8 bit Sklansky Adder : Step 2 [4]

Fig 2.8 Build 8 bit Sklansky Adder : Step 3 [4]

Fig 2.9 Build 8 bit Sklansky Adder: Step 4 [4]

Fig 3.1 CRT based R/B Converter [2]

Fig 3.2 Modulo-m adder [3]

Fig 3.3 Modified 10 Bit SQRT CSLA [8]

Fig 3.4 Truth table of single bit Full Adder, where the upper half part is the case of $C_{in}=0$ and Lower part is the case if $C_{in}=1$ [8]

Fig 3.5 Internal Structure of CSLA using Common Boolean Logic [8]

Fig 3.5 Power comparision between SQRT CLSA using excess decoders and CLSA using Common Boolean logic. [8]

Fig 3.6 Delay comparision between SQRT CLSA using excess decoders and CLSA using Common Boolean logic. [8]

Fig 4.1 Object table for input 110100101

Fig 4.2 Object table for input 110111101

Fig 4.3 Object table for input 110101101

Fig 4.4 Object table for input 110011100

Fig 4.5 Object table for input 110101111

Fig 4.6 Complete waveforms for different input

LIST OF ACRONYM

RNS Residue Number System
CRT Chinese Remainder Theorem
B/R Binary-to-Residue
R/B Residue-to-Binary
ROM Read Only Memory
LUT Look-Up Table
SQRT Square root technique
CSLA Carry Select Adder
CBL Common Boolean Logic

Software used

MODELSIM SE PLUS 6.4

Language Used

VERILOG [1]

Chapter 1

Introduction

1.1 General RNS Based Processor Structure :

A general structure of a typical RNS processor is shown in Figure 1.1. The RNS represented data is processed in parallel with no dependence or carry propagation between the processing units. The process of encoding the input data into RNS representation is called Forward Conversion, and the process of converting back the output data from RNS to conventional representation is called Reverse Conversion [2, 3].

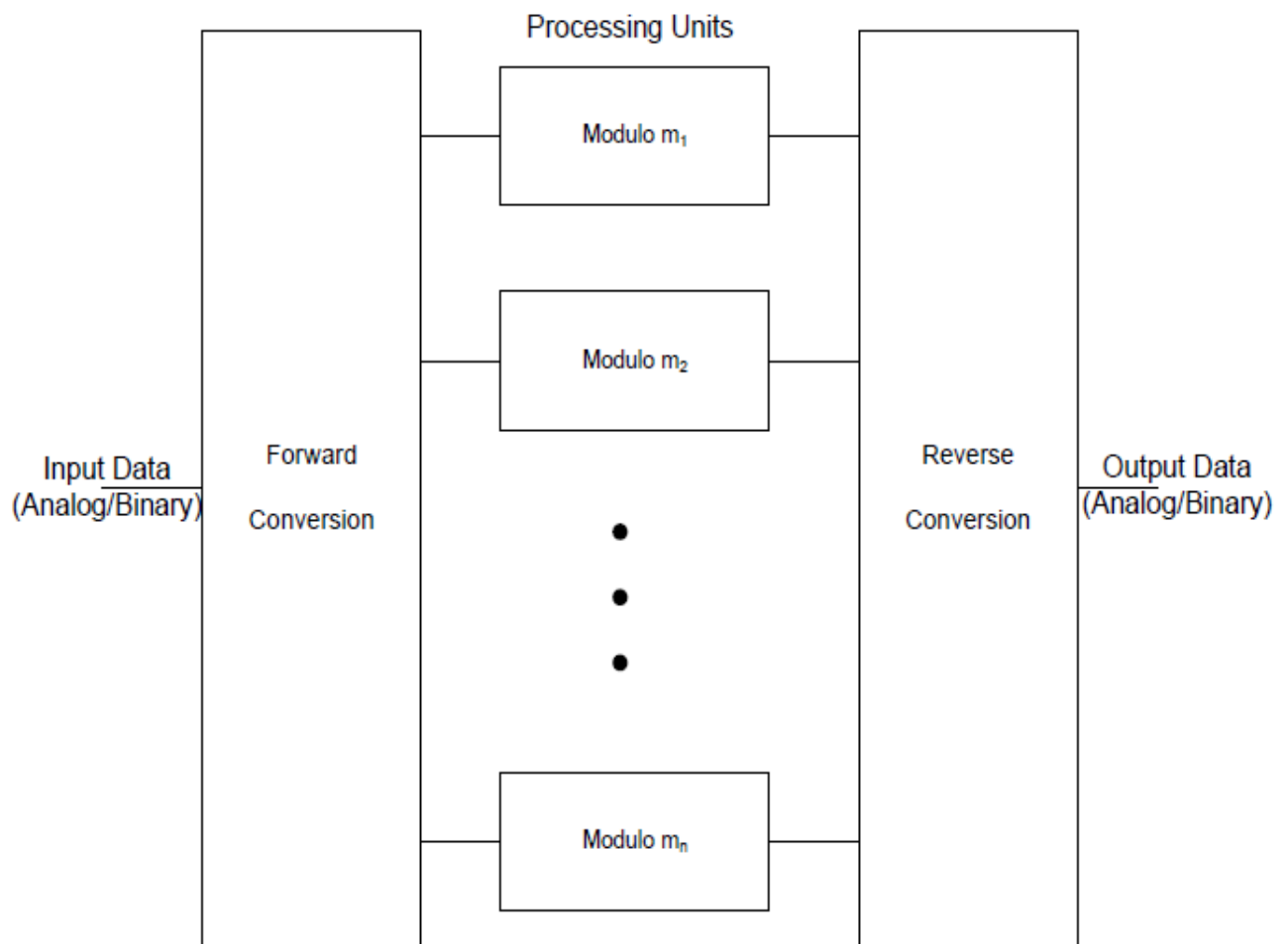


Fig1.1 General Structure Of RNS Based Processor [2]

The conversion stages are very critical in the evaluation of the performance of the overall RNS. Conversion circuitry can be very complex and may introduce latency that offsets the speed gained by the RNS processors. For a full RNS based system, the interaction with the analog world requires conversion from analog to residue and vice versa. Usually, this is done in two steps where conversion to binary is an intermediate stage. This makes the conversion stage inefficient due to their increased latency and complexity. To build an RNS processor that can replace the digital processor in a certain application; we need to develop conversion circuits. The reverse conversion process is based on the Chinese Remainder Theorem (CRT) technique [2, 3]. Investigating new conversion schemes can lead to overcoming some obstacles in the RNS implementation of different applications.

1.2 Conventional Number System:

In general, numbers may be signed, and for binary digital arithmetic there are three standard notations that have been traditionally used for the binary representation of signed numbers. These are *sign-and-magnitude*, *one's complement*, and *two's complement*. Of these three, the last is the most popular, because of the relative ease and speed with which the basic arithmetic operations can be implemented. Sign-and-magnitude notation has the convenience of having a sign-representation that is similar to that used in ordinary decimal arithmetic. And one's complement, although a notation in its own right, more often appears only as an intermediate step in arithmetic involving the other two notations. The sign-and-magnitude notation is derived from the conventional written notation of representing a negative number by prepending a sign to a magnitude that represents a positive number.

For binary computer hardware, a single bit suffices for the sign: a sign bit of 0 indicates a positive number, and a sign bit of 1 indicates a negative number. For example, the representation of the number positive-five in six bits is 000101, and the corresponding representation of negative-five is 100101. Note that the representation of the sign is independent of that of the magnitude and takes up exactly one bit; this is not the case both with one's complement and two's complement notations.

Sign-and-magnitude notation has two representations, 000...0 and 100...0, for the number zero; it is therefore redundant. Addition and subtraction are harder to implement in this notation than in one's complement and two's complement notations; and as these are the most common arithmetic operations, true sign-and-magnitude arithmetic is very rarely implemented.

In one's complement notation, the representation of the negation of a number is obtained by inverting the bits in its binary representation, that is, the 0s are changed to 1s and the 1s are changed to 0s. The leading bit again indicates the sign of the number, being 0 for a positive number and 1 for a negative number. We shall therefore refer to the most significant digit as the *sign bit*, although here the sign of a negative number is in fact represented by an infinite string of 1s that in practice is truncated according to the number of bits used in the representations and the magnitude of the number represented. It is straightforward to show that the n -bit representation of the negation of a number N is also, when interpreted as the representation of an unsigned number, that of $2^n - 1 - N$. (This point will be useful in subsequent discussions of basic residue arithmetic.) The one's complement system too has two representations for zero (00...0 and 11...1) which can be a nuisance in implementations.

1.3 Residue Number Systems :

Residue number systems are based on the *congruence* relation, which is defined as follows. Two integers a and b are said to be *congruent modulo m* if m divides exactly the difference of a and b . Thus, for example, $10 \equiv 7 \pmod{3}$, $10 \equiv 4 \pmod{3}$, $10 \equiv 1 \pmod{3}$, and $10 \equiv -2 \pmod{3}$. The number m is a *modulus* or *base*, and we shall assume that its values exclude unity, which produces only trivial congruences.

If q and r are the quotient and remainder, respectively, of the integer division of a by m —that is, $a = q.m + r$ —then, by definition, we have $a \equiv r \pmod{m}$. The number r is said to be the *residue* of a with respect to m , and we shall usually denote this by $r = |a|_m$. The set of m smallest values, $\{0, 1, 2, \dots, m-1\}$, that the residue may assume is called the set of *least positive residues modulo m* . Unless otherwise specified, we shall assume that these are the only residues in use.

Suppose we have a set, $\{m_1, m_2, \dots, m_N\}$, of N positive and pairwise relatively prime moduli¹. Let M be the product of the moduli. Then every number $X < M$ has a unique representation in the residue number system, which is the set of residues $\{|X|_{m_i} : 1 \leq i \leq N\}$. A partial proof of this is as follows. Suppose X_1 and X_2 are two different numbers with the same *residue-set*. Then $|X_1|_{m_i} = |X_2|_{m_i}$, and so $|X_1 - X_2|_{m_i} = 0$. Therefore $X_1 - X_2$ is the least common multiple (lcm) of m_i . But if the m_i are relatively prime, then their lcm is M , and it must be that $X_1 - X_2$ is a multiple of M . So it cannot be that $X_1 < M$ and $X_2 < M$. Therefore, the set $\{|X|_{m_i} : 1 \leq i \leq N\}$ is unique and may

be taken as the representation of X . We shall write such a representation in the form (x_1, x_2, \dots, x_N) , where $x_i = |X|_{m_i}$, and we shall indicate the relationship between X and its residues by writing $X \sim (x_1, x_2, \dots, x_N)$. The number M is called the *dynamic range* of the RNS, because the number of numbers that can be represented is M . For unsigned numbers, that range is $[0, M - 1]$. Representations in a system in which the moduli are not pairwise relatively prime will be not be unique: two or more numbers will have the same representation.

1.4 Mathematical Fundamentals:

The congruences are explained in details with their properties. These properties form a solid background to understand the process of conversion between the conventional system and the RNS. Basic algebra related to RNS is introduced here. This includes finding the additive and the multiplicative inverses, and some properties of division and scaling which are not easy operations in RNS.

1.4.1 Basic Definitions and Congruences

Residue of a number

The basic relationship between numbers in conventional representation and RNS representation is the following congruence:

$$X \bmod m_i = r_i$$

Where m_i is the *modulus*, and r_i is the *residue*. The residue is defined as the least positive remainder when the number X is divided by the modulus m_i .

1.4.2 Basic Algebraic Operations

Addition (or subtraction)

Addition (or subtraction) of different numbers in the RNS representation can be done by individually adding (or subtracting) the residues with respect to the corresponding moduli.

Consider the moduli-set $S = \{m_i\}$, and the numbers X and Y are given in RNS representation:

$$X = \{x_1, x_2, \dots, x_n\} \text{ and } Y = \{y_1, y_2, \dots, y_n\}$$

Then,

$$Z = X + Y = \{z_1, z_2, \dots, z_n\}$$

$$\text{where } z_i = (x_i + y_i) \bmod m_i$$

This property can be applied to subtraction as well, where subtraction of Y from X is considered as the addition of \bar{Y} . The modulo operation is distributive over addition (and subtraction):

$$|X \mp Y|_m = ||X|_m \mp |Y|_m|_m$$

Multiplication

Multiplication in RNS can be carried out by multiplying the individual residues with respect to the corresponding moduli. Consider the moduli-set $S = \{m_1, m_2, \dots, m_n\}$, and the numbers X and Y are given in RNS representation.

$$X = \{x_1, x_2, \dots, x_n\} \quad \text{and} \quad Y = \{y_1, y_2, \dots, y_n\}$$

Then,

$$Z = X \times Y = \{z_1, z_2, \dots, z_n\}$$

where $z_i = (x_i \times y_i) \bmod m_i$

The modulo operation is distributive over multiplication:

$$|X \times Y|_m = ||X|_m \times |Y|_m|_m$$

Additive Inverse

The relation between the residue r_i and its additive inverse \bar{r}_i is defined by the congruence:

$$(r_i + \bar{r}_i) \bmod m_i = 0$$

The additive inverse \bar{r}_i can be obtained using the following operation:

$$\bar{r}_i = (m_i - r_i) \bmod m_i$$

Subtraction is one application of this property, where subtraction is regarded as the addition of the additive inverse.

Multiplicative Inverse

The multiplicative inverse r_i^{-1} of the residue r_i is defined by the congruence:

$$(r_i \times r_i^{-1}) \bmod m_i = 1 \quad \text{-----(1.1)}$$

where r_i^{-1} exists only if r_i and m_i are relatively prime.

There is no general method of obtaining the multiplicative inverse. The multiplicative inverse is usually obtained by brute-force search.

1.5 Conversion:

The most direct way to convert from a conventional representation to a residue one, a process known as *forward conversion*, is to divide by each of the given moduli and then collect the remainders. This, however, is likely to be a costly operation if the number is represented in an arbitrary radix and the moduli are arbitrary. If, on the other hand, the number is represented in radix-2 (or a radix that is a power of two) and the moduli are of a suitable form (e.g. 2^n-1), then there are procedures that can be implemented with more efficiency. The conversion from residue notation to a conventional notation, a process known as *reverse conversion*, is more difficult (conceptually, if not necessarily in the implementation) and so far has been one of the major impediments to the adoption of RNS. One way in which it can be done is to assign weights to the digits of a residue representation and then produce a “conventional” (i.e. positional, weighted) mixed-radix representation from this. This mixed-radix representation can then be converted into whatever conventional form is desired. In practice, the use of a direct conversion procedure for the latter can be avoided by carrying out the arithmetic of the conversion in the notation for the result. Another approach involves the use of the Chinese Remainder Theorem, which is the basis for many algorithms for conversion from residue to conventional notation.

Chapter 2

Conversion between Binary and RNS Representation

2.1 Forward Conversion from Binary to RNS Representation

The forward conversion stage is of paramount importance as it is considered as an overhead in the overall RNS. Choosing the most appropriate scheme depends heavily on the used modulus set. Forward converters are usually classified based on the used moduli into two categories. The first category includes forward converters based on arbitrary modulus sets. These converters are usually built using look-up tables. The second category includes forward converters based on special modulus sets. The use of special modulus sets simplifies the forward conversion algorithms and architectures. The special modulus set converters are usually realized using pure combinational logic. Forward conversion based on the special modulus set $\{2^n - 1, 2^n, 2^n + 1\}$ [2, 3], how complexity of the overall design is minimized which reduces the overhead introduced by the forward converter, some architectures for implementing the modulo addition that are used in the realization of all forward converters are represented in this report.

2.2 Parallel Conversion

Forward conversion from binary to RNS representation can be obtained parallel. Suppose X is partitioned into k blocks, each of p -bits. Let X be partitioned into the blocks $B_{k-1}B_{k-2} \dots B_1B_0$, then:

$$X = \sum_{j=0}^{k-1} 2^{jB} B_j$$

$$|X|_m = \left| \sum_{j=0}^{k-1} 2^{jB} B_j \right|_m = \left| \sum_{j=0}^{k-1} |2^{jB} B_j|_m \right|_m \text{-----A}$$

Equation ‘A’ can be directly implemented by storing the values $|2^{jB} B_j|_m$ in k look-up tables, where k is the number of partitioning blocks [2]. The values of B_j are used to address the

values $|2^j B_j|_m$ in the look-up table (LUT). These values are then added using a multi-operand modulo adder. A typical implementation of Equation 'A' is shown in Figure 2.1.

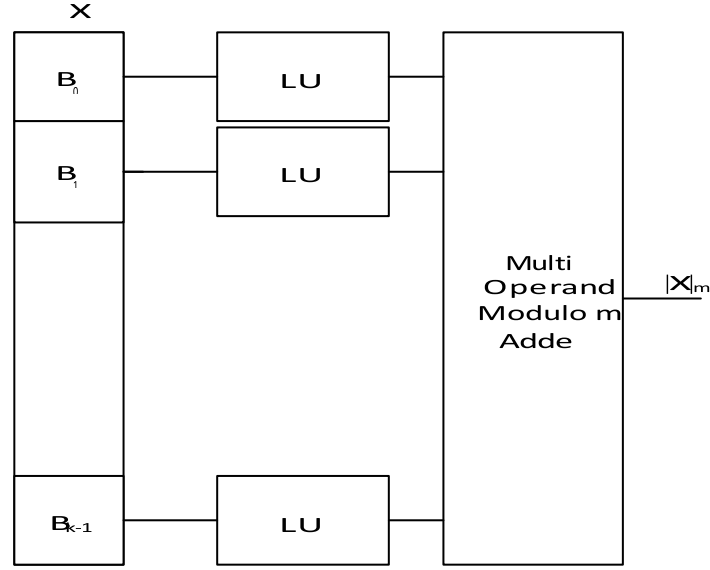


Figure 2.1. Parallel forward converter [3]

Each look-up table (LUT) is a ROM cell that has a size of $(p \times \log_2 m)$ bits, where p is the number of bits in each block, and m is the modulus. Parallel forward converters are faster and more adequate for high speed applications. However, the parallel converters require k look-up tables and a modulo adder that adds k operands with respect to modulus m .

2.3 Special Moduli-Set Forward Converter

Choosing a special moduli-set is the preferred choice to facilitate and expedite the conversion stages. The special moduli-set forward converters are the most efficient available converters in terms of speed, area, and power. Usually, the special moduli-sets are referred to as 'low-cost moduli-sets'. **The special moduli-set $\{2^n - 1, 2^n, 2^n + 1\}$ for $n=3$ is represented in this project [2, 3].**

The special moduli-set converters are usually implemented using pure combinational logic. To compute the residue of a number X (in binary representation) with respect to modulus m where m is restricted to $\{2^n - 1, 2^n, 2^n + 1\}$. Derivation of formulas that facilitate the algorithm used to obtain the residues is shown below.

2.31 The Special Moduli-Set $\{2^n - 1, 2^n, 2^n + 1\}$

A general algorithm is presented to convert X (in binary representation) into RNS representation with respect to the special moduli-set $\{2^n - 1, 2^n, 2^n + 1\}$ [2, 3]. We first partition X into 3 blocks, each of n bits. B_1, B_2 and B_3 where these blocks can be represented as follows:

$$\begin{aligned} B_1 &= \sum_{j=2n}^{3n-1} x_j 2^{j-2n} \\ B_2 &= \sum_{j=n}^{2n-1} x_j 2^{j-n} \\ B_3 &= \sum_{j=0}^{n-1} x_j 2^j \end{aligned}$$

Thus,

$$X = B_1 2^{2n} + B_2 2^n + B_3$$

The residue r_2 is the easiest to compute. The n least significant bits constitute the remainder when X is divided by 2^n . Hence r_2 is the number represented by the least significant n bits of X . These bits are obtained by shifting to the right by n bits.

In order to determine the residues, r_1 and r_3 , we first partition X into three n -bit blocks, $\mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3$. The residue r_1 is then obtained as

$$\begin{aligned} r_1 &= |X|_{2^n+1} \\ &= |\mathbf{B}_1 2^{2n} + \mathbf{B}_2 2^n + \mathbf{B}_3|_{2^n+1} \\ &= \left| |\mathbf{B}_1 2^{2n}|_{2^n+1} + |\mathbf{B}_2 2^n|_{2^n+1} + |\mathbf{B}_3|_{2^n+1} \right|_{2^n+1} \end{aligned}$$

\mathbf{B}_3 is an n -bit number and therefore is always less than 2^n+1 ; so its residue is simply the binary equivalent of this term. The residues of the other two sums are computed as

$$|\mathbf{B}_1 2^{2n}|_{2^n+1} = \left| |\mathbf{B}_1|_{2^n+1} |2^{2n}|_{2^n+1} \right|_{2^n+1}$$

and

$$|\mathbf{B}_2 2^n|_{2^n+1} = \left| |\mathbf{B}_2|_{2^n+1} |2^n|_{2^n+1} \right|_{2^n+1}$$

Each of \mathbf{B}_1 and \mathbf{B}_2 is represented in n bits and there must be less than $2^n + 1$. And the residue of 2^{2n} with respect to $2^n + 1$ is

$$\begin{aligned}
|2^{2n}|_{2^{n+1}} &= |2^n 2^n|_{2^{n+1}} \\
&= |2^n + 1 - 1|_{2^{n+1}} |2^n + 1 - 1|_{2^{n+1}} \\
&= -1 \times -1 \\
&= 1
\end{aligned}$$

It follows from this that the residue of 2^n with respect to $2^n + 1$ is -1 . Therefore,

$$r1 = |\mathbf{B}1 - \mathbf{B}2 + \mathbf{B}3|_{2^{n+1}}$$

Similarly, to compute $r3$, we first observe that

$$\begin{aligned}
|2^{2n}|_{2^{n-1}} &= |2^n - 1 + 1|_{2^{n-1}} \times |2^n - 1 + 1|_{2^{n-1}} \\
&= 1 \times 1 \\
&= 1
\end{aligned}$$

Also, $|2^n|_{2^{n-1}}$ is 1. So

$$r3 = |\mathbf{B}1 + \mathbf{B}2 + \mathbf{B}3|_{2^{n-1}}$$

From the above, we may surmise three modular adders will suffice for the computation of the residues. If the magnitudes of the numbers involved are small, as will be the case for small moduli, the complexity of the overall conversion will not be high.

A typical architecture for the implementation of a forward converter from binary to RNS representation for the special moduli-set $\{2^n - 1, 2^n, 2^n + 1\}$ is shown in Figure 2.2.

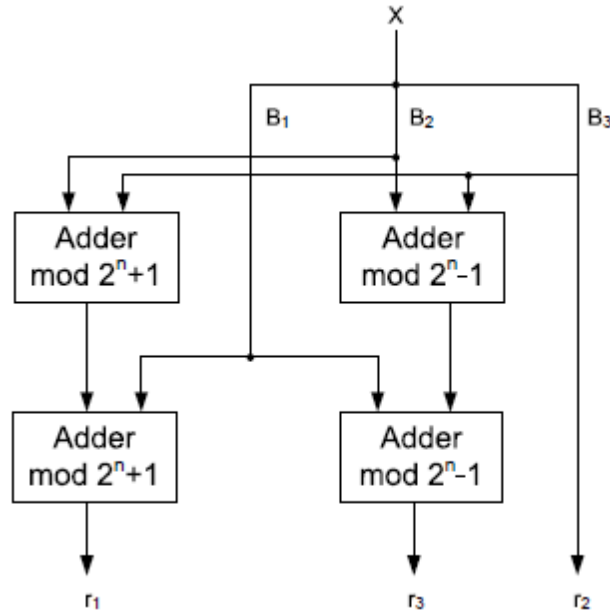


Fig 2.2 $\{2^n - 1, 2^n, 2^n + 1\}$ forward converter [3]

2.4 Modulo Adder For Special Moduli :

2.41 Modulo 2^n Adder

Modulo 2^n addition is the easiest modulo addition operation in the residue domain because it does

not require any extra overhead compared to the conventional addition. Modulo 2^n addition of any

two numbers X and Y , each of n bits, is done by adding the two numbers using a conventional adder. The result is an $n + 1$ bit output, where the most significant bit is the carryout. The residue

is the first n lowest significant bits, and the final carry-out is neglected. Therefore, modulo 2^n addition is the most efficient modulo addition operation in the residue domain.

2.42 Modulo $2^n - 1$ Adder

The modulo $2^n - 1$ adder is an important arithmetic unit in RNS because $2^n - 1$ is commonly used modulus in most special moduli-sets, e.g. $\{2^n - 1, 2^n, 2^n + 1\}$. CLA architecture to implement the $2^n - 1$ modulo addition will be represented [3]. Here, we shall present the basic idea behind these algorithms and architectures.

To understand the operation of modulo $2^n - 1$ addition of any two numbers X and Y , where $0 \leq X, Y < m$, we need to distinguish between three different cases:

- a) $0 \leq X + Y < 2^n - 1$
- b) $X + Y = 2^n - 1$
- c) $2^n - 1 < X + Y < 2^{n+1} - 2$

In the first case, the result of the conventional addition is less than the upper limit $2^n - 1$ and no carry-out (C_{out}) is generated at the most significant bit. In this case, the modulo addition of X and Y is equivalent to the conventional addition. In the second case, the result is equal to $2^n - 1$ (i.e. all 1's in binary representation). However, from RNS definition, the result has to be less than $2^n - 1$. In this case, the result should be zero. This case can be detected when all bits of the resulting number are ones (i.e. all $P_i = x_i \oplus y_i$ are ones). Correction is done simply in this case by adding a one and neglecting the carry-out. In the third case, the result of the conventional addition exceeds $2^n - 1$ and a carry-out is generated at the most significant

bit. This case is easily detected by the carry-out. Correction is done by ignoring the carry-out (equivalent to subtracting 2^n) and adding 1 to produce the correct result.

Carry Look Ahead Adder Structure for modulo $2^n - 1$ Adder

The logic equations for such an adder are

$$\begin{aligned} G_i &= A_i B_i \\ P_i &= \overline{A_i} B_i + A_i \overline{B_i} = A_i \oplus B_i \\ C_i &= G_i + P_i C_{i-1} \\ S_i &= P_i \oplus C_{i-1} \end{aligned}$$

and unwinding the carry equation yields

$$C_0 = G_0 + P_0 C_{-1}$$

$$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{-1} \quad C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1}$$

.

$$C_i = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} P_{i-2} \dots P_0 C_{-1}$$

where C_{-1} is the carry-in to adder, i.e. into the least significant bit-position. Suppose such an adder is used to perform residue addition in two cycles. The first cycle would consist of the addition of A and B , with $C_{-1} = 0$. And in the second cycle $C_{-1} = C_{\text{out}}$, where C_{out} is the carry-out from the first addition. Consider now the least significant bit-slice. Let C_i^q denote the carry from bit-slice i during the cycle q , let $S'_{n-1} S'_{n-2} \dots S'_0$ denote the (intermediate) sum produced in that cycle, and let G_i and P_i be the carry generate and propagate signals during the addition (in the first cycle) of A and B . In the first cycle, since $C_{-1}^1 = 0$, and there is a carry from bit-slice 0 only if one is *generated* there; that is, $C_0^1 = G_0$. In the second cycle, the operands into that bit slice are S'_0 , an implicit 0, and C_{-1} (formerly C_{n-1}^1). In that cycle a carry cannot be generated from bit-slice 0, but C_{-1} may be *propagated*; and the propagation signal in that case is $S'_0 \oplus 0$, which is just S'_0 . Now, $S'_0 = P_0$ (since $C_{-1}^1 = 0$; so $C_{-1}^2 = P_0 C_{n-1}^1$).

Note that C_0^1 and C_0^2 are independent, and the expression for a carry in either cycle is $C_0^1 + C_0^2 = G_0 + P_0 C_{n-1}^1$, which is exactly the normal expression for a carry from that position (Equation 4.7). In general, for a given bit-slice, the generation and propagation of carries are mutually exclusive and occur in different cycles. That is, if in the first cycle a carry is generated at bit-slice i , then that carry, if it propagates to the end of the adder and “wraps” around, cannot propagate beyond bit-slice i . This mutual exclusivity of G_i and P_i signals may be similarly applied to all the other bit-slices and combining this with the elimination of C_{-1} (by substituting its definition) leads to a residue adder which requires only a little more logic.

The logic equations for a modulo-7 adder ($n = 3$) with two representations for zero are :

$$\begin{aligned}
G_i &= A_i B_i & i = 0, 1, 2 \\
P_i &= A_i \oplus B_i \\
C_{-1} &= C_2 \\
C_0 &= G_0 + G_2 P_0 + P_2 G_1 P_0 \\
C_1 &= G_1 + P_1 G_0 + G_2 P_1 P_0 \\
C_2 &= G_2 + P_2 G_1 + P_2 P_1 G_0 \\
S_i &= P_i \oplus C_{i-1}
\end{aligned}$$

The modulo carry-lookahead adder allows two representations for zero in the result: $00\cdots 0$ and $11\cdots 1$ (i.e. $2^n - 1$). To modify it so that only one representation is permissible, it is necessary to detect the latter and ensure that a sum of zero is instead produced. The reader can verify that the correct output will be produced if the equation above for the sum bits is changed from $S_i = P_i \oplus C_{i-1}$ to,

$$S_i = (P_i \cdot \overline{P_0^{n-1}}) \oplus C_{i-1}$$

If $P_0^{n-1} = 0$, then this equation reduces to $S_i = P_i \oplus C_{i-1}$, which is the correct formulation for an output other than $11\cdots 1$. If $P_0^{n-1} = 1$, an output of $11\cdots 1$ needs to be converted into $00\cdots 0$. In this case the equation reduces to $S_i = C_{i-1}$. This is evidently correct, since $P_i = 1$, for all i , means that no carry is generated anywhere; and, as then $C_{-1} = 0$, there is also no carry to propagate anywhere. Therefore, for all i , $C_i = 0$ and so $S_i = 0$. The design of the modified residue adder is shown below.

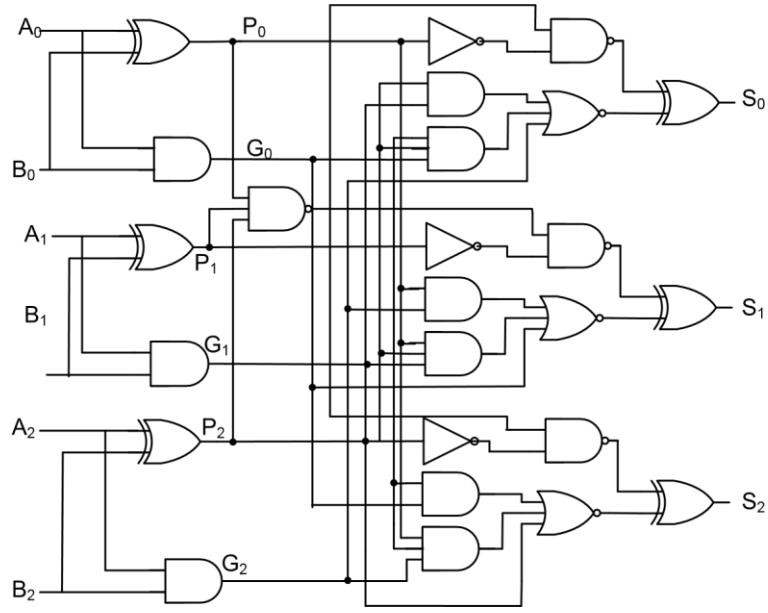


Fig 2.3 Modulo $2^n - 1$ carry look ahead adder [3]

2.43 Modulo $2^n + 1$ Adder

The modulo $2^n + 1$ adder is the bottleneck of the design of a forward converter from binary to RNS representation for the special moduli-set $\{2^n - 1, 2^n, 2^n + 1\}$ [3]. Its importance arises from the fact that designing an efficient modulo $2^n + 1$ adder is more difficult than that of the other two moduli. This is due to difficulties in detecting when the result is equal to $2^n + 1$ and when it exceeds $2^n + 1$.

In a similar way to that used in modulo $2^n - 1$ addition, three cases have to be distinguished [4]. First, we define Z as follows:

$$Z = X + Y - (2^n + 1)$$

Then, we define the three cases as follows:

- a) $X + Y \geq 2^n + 1$ (i.e. $Z \geq 0$)
- b) $X + Y = 2^n$ (i.e. $Z = -1$)
- c) $X + Y < 2^n + 1$ and $X + Y \neq 2^n$ (i.e. $Z < 0$ but $Z \neq -1$)

In the first case, $(X + Y) \bmod (2^n + 1)$ is simply equal to Z . In the second case,

$(X + Y) \bmod (2^n + 1)$ is obtained from Z by setting the most significant bit of Z to 1 and adding 1 to the result. In the third case, Z is negative, and $(X + Y) \bmod (2^n + 1)$ is obtained from Z by

setting the most significant bit to 0 and adding 1 to the result. In summary:

$$|X + Y|_{2^n+1} = \begin{cases} Z & : Z \geq 0 \\ 2^n + |Z + 1|_{2^n} & : Z = -1 \\ |Z + 1|_{2^n} & : otherwise \end{cases}$$

A possible architecture for implementing a modulo $2^n + 1$ adder is proposed in [4]. The architecture is shown in Figure 2.7. A carry-save adder (CSA) reduces the three inputs X , Y , and $-(2^n + 1)$ to two: partial sum (\check{S}) and partial carry (\check{C}) [3, 7]. The two values \check{S} and \check{C} are then processed using a parallel-prefix adder [4]. Case (b) is detected if $P_0^n = P_0 P_1 \dots P_n = 1$. Then, the correction is done by adding P_0^n as an end-around carry and setting $S_n = P_0^n$. Case (c) is detected if C_{n-1} and therefore C_n is 0. The correction is done in this case by adding the inverse of the end-around carry $\overline{C_n}$ and setting S_n to zero.[2, 3]

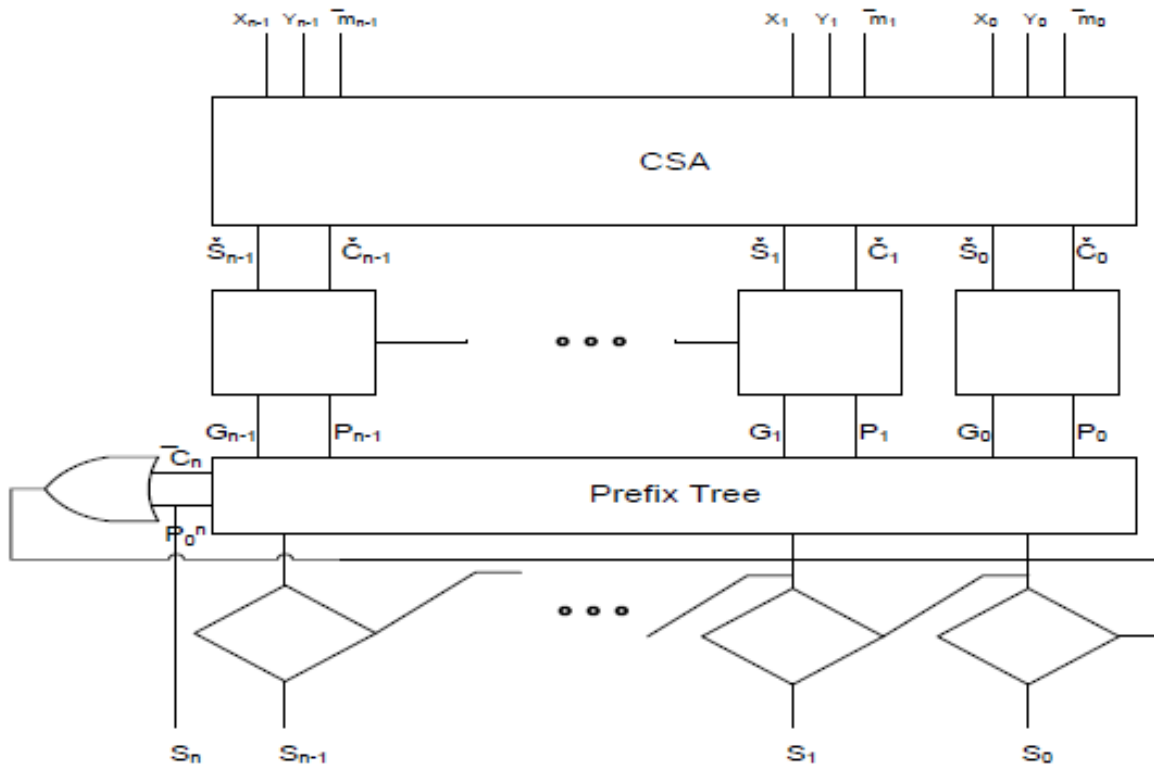


Fig 2.4 Modulo $2^n + 1$ Adder [2]

Parallel Prefix Adder

Parallel prefix tree used as intermediate in modulo $2^n + 1$ adder can be implemented using Sklansky Adder whose algorithm will be described later. Before implementing Sklansky adder, basic algorithm of parallel prefix adder should be known [4]. The idea is to compute small group of intermediate prefixes and then find large group prefixes, until all the carry bits are computed. These adders have tree structures within a carry-computing stage similar to the carry propagate adder. However, the other two stages for these adders are called pre-computation post-computation stages.

In pre-computation stage, each bit computes its carry generate/propagate and temporary sum as shown below,

$$\begin{aligned}g_i &= a_i \cdot b_i \\p_i &= a_i \text{ (xor) } b_i\end{aligned}$$

In the prefix stage, the group carry generate/propagate signals are computed to form the carry chain and provide the carry-in for the adder below is repeated here as a reminder

$$\begin{aligned}G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:j} \\P_{i:j} &= P_{i:k} \cdot P_{k-1:j}\end{aligned}\tag{2.2}$$

In the post-computation stage, the sum and carry-out are finally produced. The carry-out can be omitted if only a sum needs to be produced.

$$\begin{aligned}s_i &= t_i \text{ (xor) } G_{i-1} \\c_{out} &= g_{n-1} + p_{n-1} \cdot G_{n-2:-1}\end{aligned}$$

where $G_{i-1} = C_i$ with the assumption $g_{-1} = c_{in}$. The general diagram of parallel-prefix structures is shown in Figure 2.5, where an 8-bit case is illustrated.

Equation (2.2) can be represented as **Black Cell**. Calculating only $G_{i:j}$ is represented by **Grey Cell**.

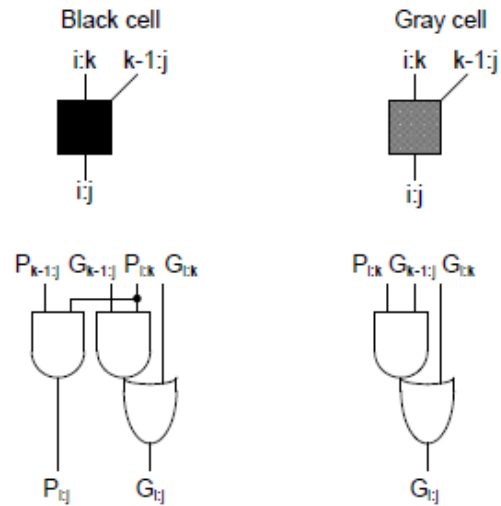


Fig 2.5 Cell Definition [4]

Sklansky Prefix Tree :

The inputs $g_i=p_i$ go from the top and the outputs c_i are at the bottom. The LSB is labeled as -1 where the carry-input (c_{in}) locates. The objective is to obtain all c_i 's in the form of $G_{i-1:-1}$'s, where $c_0 = G_{-1:-1}$; $c_1 = G_{0:-1}$; $c_2 = G_{1:-1}$; \dots ; $c_{n-1} = G_{n-2:-1}$

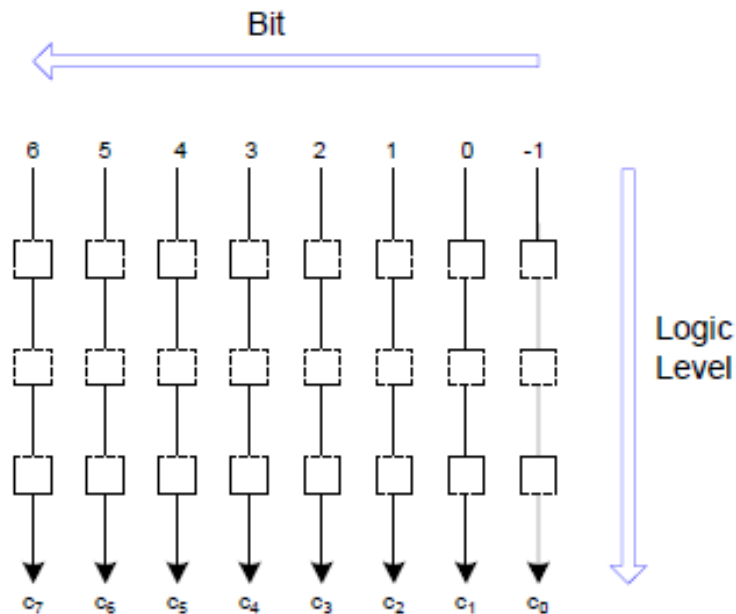


Fig 2.6 Build 8 bit Sklansky Adder : Step 1 [4]

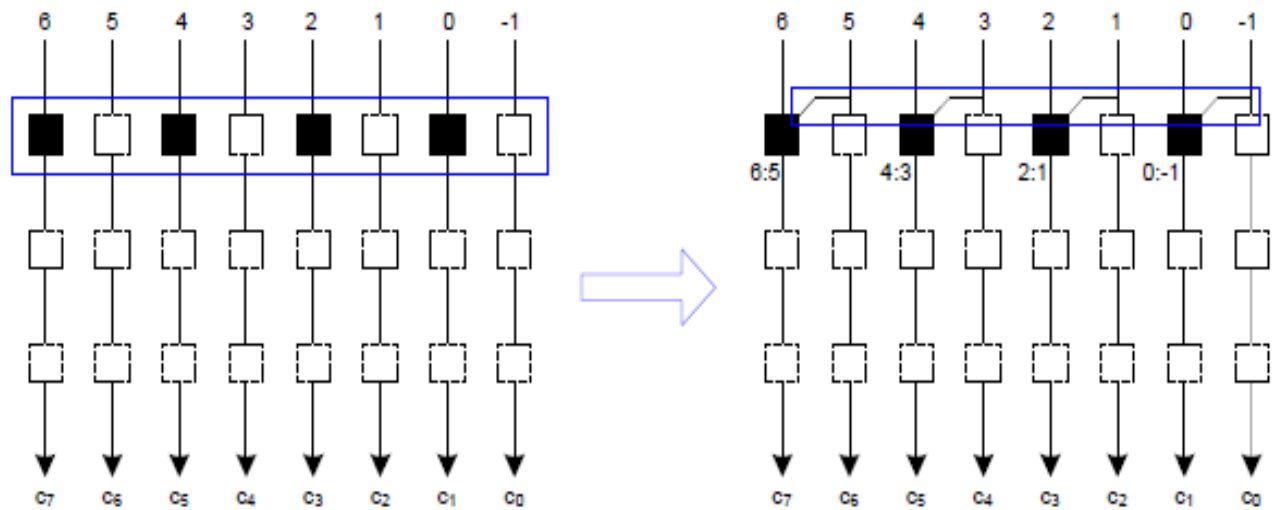


Fig 2.5 Build 8 bit Sklansky Adder : Step 2 [4]

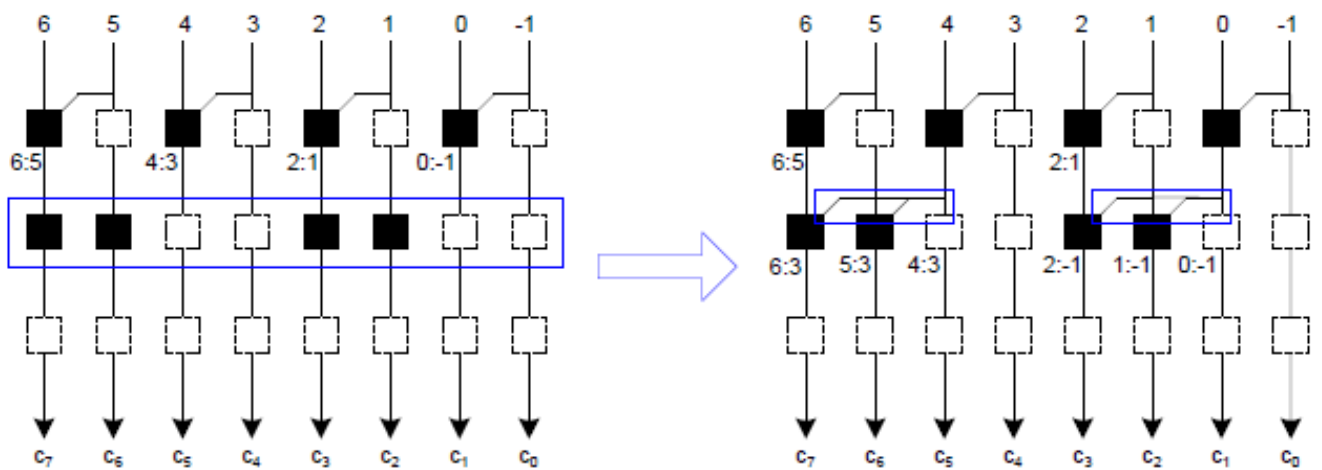


Fig 2.7 Build 8 bit Sklansky Adder : Step 3 [4]

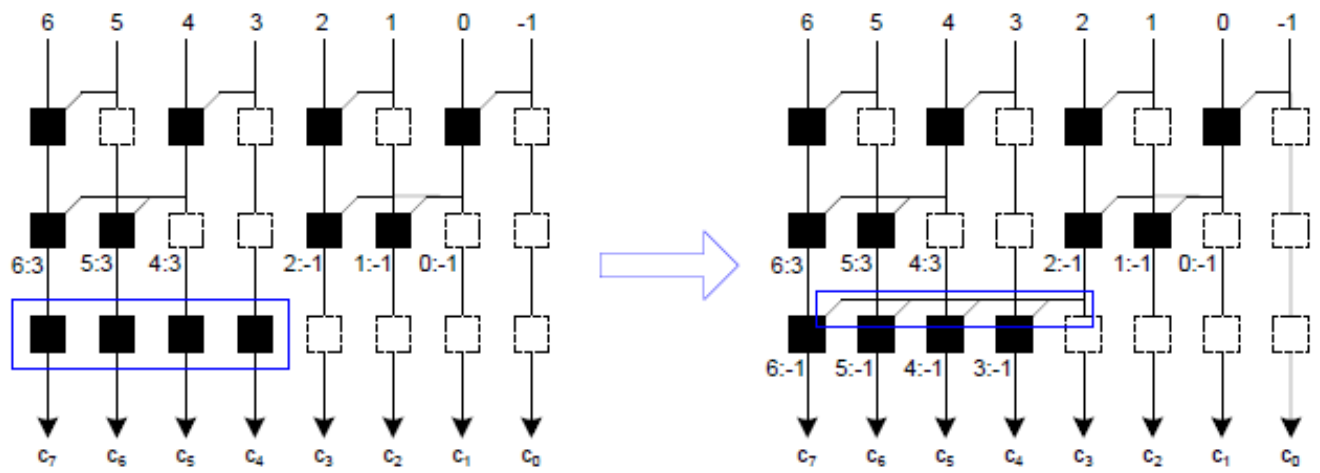


Fig 2.8 Build 8 bit Sklansky Adder: Step 4 [4]

Chapter 3

Reverse Conversion from RNS to Binary Representation

3.1 Introduction

Reverse conversion algorithms in the literature are all based on either Chinese Remainder Theorem (CRT) or Mixed-Radix Conversion (MRC) [2]. The MRC is an inherently sequential approach. On the other hand, the CRT can be implemented in parallel [2]. The reverse conversion is one of the most difficult RNS operations and has been a major, if not the major, limiting factor to a wider use of RNS. In general, the realization of a VLSI implementation of R/B converters is still complex and costly. Here, we derive the mathematical foundations of the CRT, and then we present possible implementations of this method in reverse conversion.

3.2 Chinese Remainder Theorem

The statement of the Chinese Remainder Theorem (CRT) is as follows:

Given a set of pair-wise relatively prime moduli $\{m_1, m_2, \dots, m_n\}$ and a residue representation $\{r_1, r_2, \dots, r_n\}$ in that system of some number X , i.e. $r_i = |X|_{m_i}$, that number and its residues are related by the equation [2, 3]:

$$|X|_M = \left| \sum_{i=1}^n r_i |M_i^{-1}|_{m_i} M_i \right|_M \text{-----(3.1)}$$

where M is the product of the m_i 's, and $M_i = M/m_i$. If the values involved are constrained so that the final value of X is within the dynamic range, then the modular reduction on the lefthand side can be omitted.

To understand the formulation of Equation (3.1), we rewrite X as:

$$\begin{aligned} X &\triangleq \{r_1, r_2, \dots, r_n\} \\ &\triangleq \{r_1, 0, \dots, 0\} + \{0, r_2, \dots, 0\} + \dots + \{0, 0, \dots, r_n\} \\ &\triangleq X_1 + X_2 + \dots + X_n \end{aligned}$$

Hence, the reverse conversion process requires finding X_i 's. The operation of obtaining each X_i is a reverse conversion process by itself. However, it is much easier than obtaining X .

Consider now that we want to obtain X_i from $\{0, 0, \dots, r_i, \dots, 0, 0\}$. Since the residues of X_i are

zeros except for r_i . This dictates that X_i is a multiple of m_j where $j \neq i$. Therefore, X_i can be expressed as:

$$X_i \triangleq r_i \times \{0, 0, \dots, 1, \dots, 0, 0\} \triangleq r_i \times \tilde{X}_i$$

where \tilde{X}_i is found such that $|\tilde{X}_i|_{m_i} = 1$. We recall from equation (1.1) that the relation between the number r_i and its inverse r_i^{-1} is as follows:

$$(r_i \times r_i^{-1}) \bmod m_i = 1$$

We define M_i as M/m_i , where $M = \prod_{i=1}^n m_i$. Then:

$$|M_i^{-1}|_{m_i} M_i \big|_{m_i} = 1$$

Since all m_i 's are relatively prime, the inverses exist:

$$\tilde{X}_i = |M_i^{-1}|_{m_i} M_i$$

and

$$\begin{aligned} X_i &= r_i \tilde{X}_i = r_i |M_i^{-1}|_{m_i} M_i \\ X &= \sum_{i=1}^n X_i = \sum_{i=1}^n r_i |M_i^{-1}|_{m_i} M_i \end{aligned}$$

To ensure that the final value is within the dynamic range, modulo reduction has to be added to both sides of the equation. The result is Equation (3.1).

Since there is no general method to obtain M_i^{-1} using Equation (1.15), the best way to implement it is to save the constants $X_i = |M_i^{-1}|_{m_i} M_i$ in a ROM[2]. These constants are then multiplied with the residues (r_i) and added using a modulo M adder [3]. This is a straightforward implementation of Equation (2.30). The resulting architecture has two main drawbacks when the dynamic range is large: one, large or many multipliers are required to multiply the constants X_i by the residues; two, a large modulo M adder is required at the final stage. One possible remedy to obviate the delay and the cost of large or many multipliers is to replace them with ROMs (look-up tables). All possible values of $r_i X_i$ are stored in the ROMs.

In this project we have represented Reverse converter for moduli set $\{2^n, 2^n - 1, 2^n + 1\}$ and $n=3$. So our modulo set becomes (7, 8, 9). Hence only three Rom's are required with modulo-504-adder to implement reverse converter.

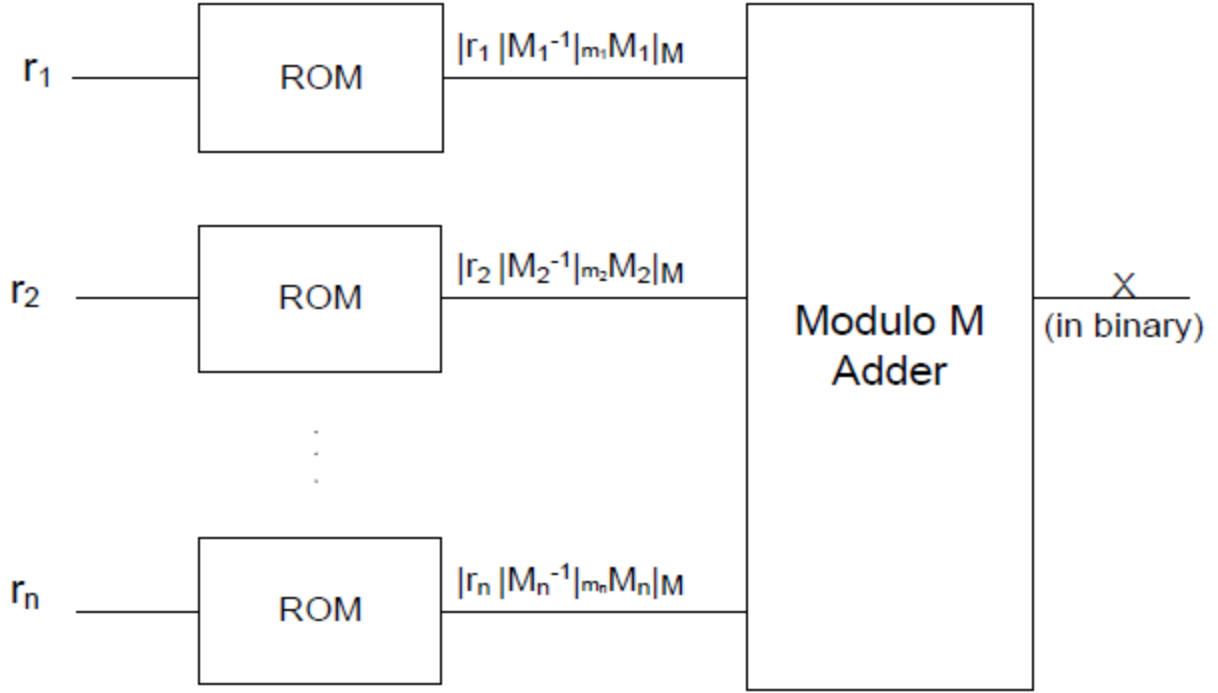


Fig 3.1 CRT based R/B Converter [2]

3.21 Modulo-m-adder

The basic idea of modulo addition of any two numbers X and Y with respect to an arbitrary modulus m is based on the following relation,

$$|X + Y|_m = \begin{cases} X + Y & : X + Y < m \\ X + Y - m & : X + Y \geq m \end{cases} \text{-----}(3.2)$$

Where $0 \leq X, Y < m$.

A typical straightforward implementation of Equation (3.2) is shown in Figure 3.2. The addition of X and Y is performed using a conventional adder. This results in an intermediate

value S . Another intermediate value $S - m$ is computed using another conventional adder[6]. Subtracting m is performed easily by adding m 's compliment (\overline{m}). In binary representation, \overline{m} also represents the value $2^n - m$. If $X + Y < m$, then $X + Y + \overline{m} < 2^n$, and the carry-out (C_{out}) is equal to 0. If $X + Y \geq m$, then $X + Y + \overline{m} = (X + Y - m) + 2^n$, and since $X + Y - m \geq 0$, a carry-out propagates in this case. The value of C_{out} instructs the multiplexer (MUX) to select the proper value between S and $S - m$.

Here in this project $m=504$.

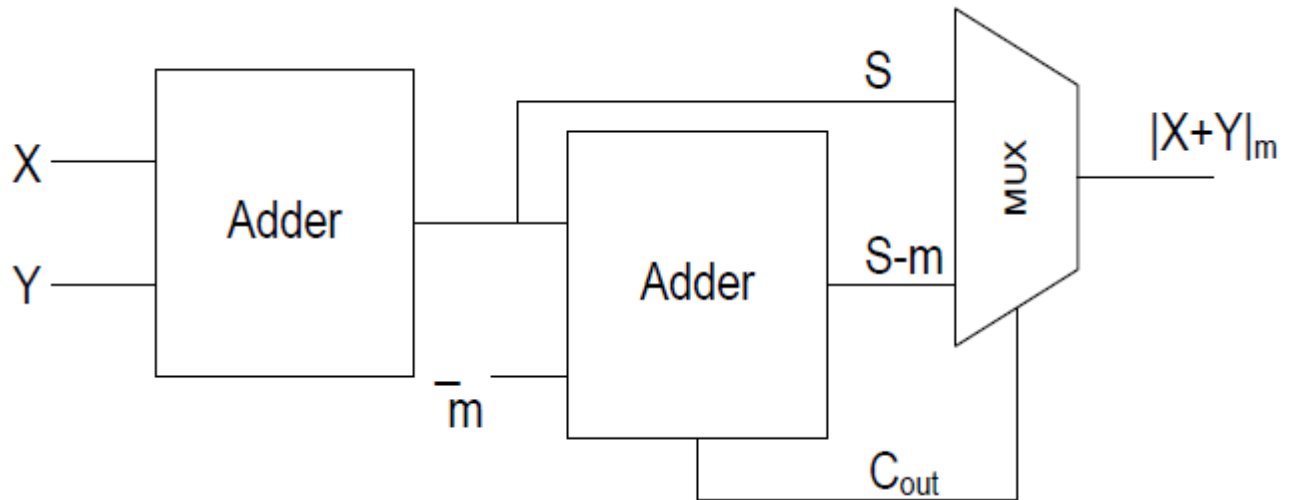


Fig 3.2 Modulo-m adder [2]

3.22 Carry Select Adder Used as subblock in modulo-m-adder

Carry Select Adder using Excess Decoders:

Carry select adder uses two stages of RCA i.e., to compute addition of bits with 0 as C_{in} and 1 as C_{in} . The conventional CSLA is area consuming due to the use of dual RCA's. The basic idea of this work is to use Binary to Excess-1 converter (BEC) instead of RCA with $C_{in}=1$ in conventional CSLA in order to reduce the area and power [8]. BEC uses less number of logic gates than N-bit full adder structure. To replace N-bit RCA, an N+1 bit BEC is required. Therefore, Modified CSLA has low power and less area than conventional CSLA. Conventional CSLA has one main disadvantage of high area usage. This advantage can be overcome in Regular Sqrt CSLA. So Sqrt CSLA is improved version of Conventional CSLA. Time delay of conventional CSLA can be decreased by having one more input into each set of adders than in previous set [8]. Regular Sqrt CSLA uses RCA's. Modified Sqrt CSLA algorithm and BEC performs the operation as that of the replaced RCA with $C_{in}=1$. Fig 3.3 shows the block diagram of modified Sqrt CSLA. This structure consumes less area, delay and power than regular Sqrt CSLA because of less number of transistors are used.

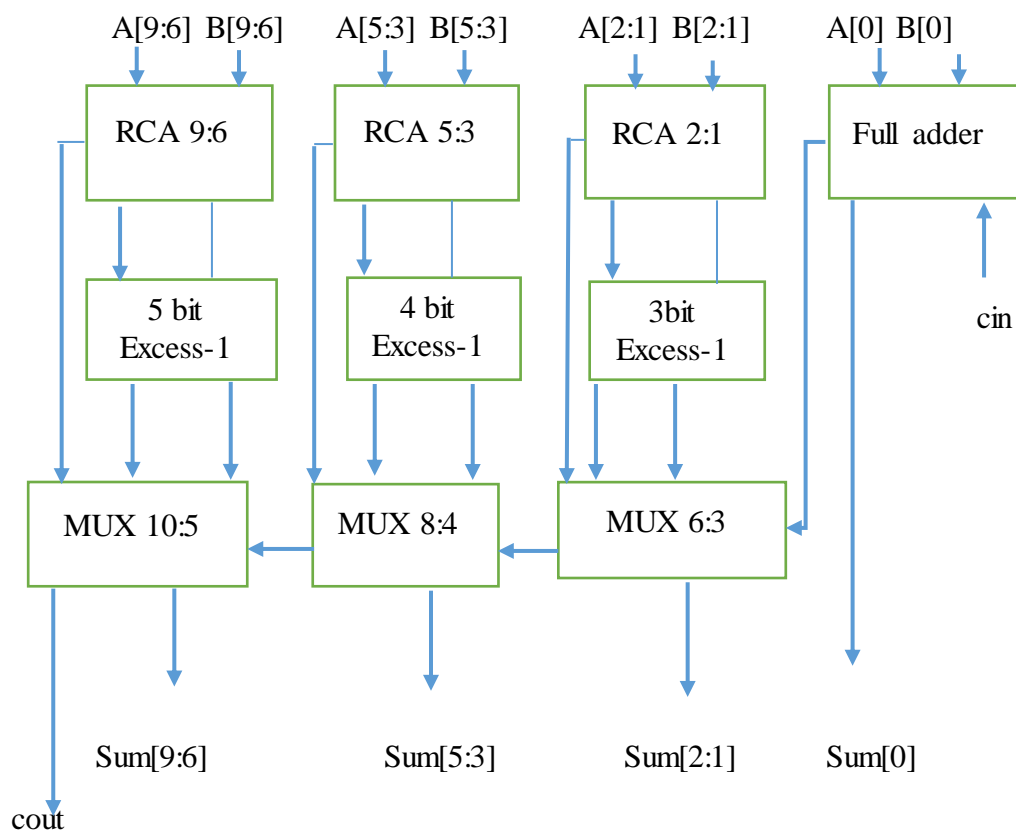


Fig 3.3 Modified 10 Bit Sqrt CSLA

Carry Select Adder using Common Boolean Logic

More efficient Modified Sqrt CSLA can be implemented using Common Boolean Logic. To remove the duplicate adder cells in the conventional CSLA, an area efficient Sqrt CSLA is proposed by sharing Common Boolean Logic (CBL) term [8]. While analysing the truth table of single bit full adder, results shows that the output of summation signal as carry-in signal is logic "0" is inverse signal of itself as carry-in signal is logic "1". It is illustrated by red circles in fig 3.4. To share the Common Boolean Logic term, we only need to implement a XOR gate and one INV gate to generate the summation pair. And to generate the carry pair, we need to implement one OR gate and one AND gate. In this way, the summation and carry circuits can be kept parallel.

Cin	A	B	S0	C0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig 3.4 Truth table of single bit Full Adder, where the upper half part is the case of Cin=0 and Lower part is the case if Cin=1 [8]

This method replaces the Binary to Excess-1 converter add one circuit by common Boolean logic. As compared with modified Sqrt CSLA, the proposed structure is little bit faster.

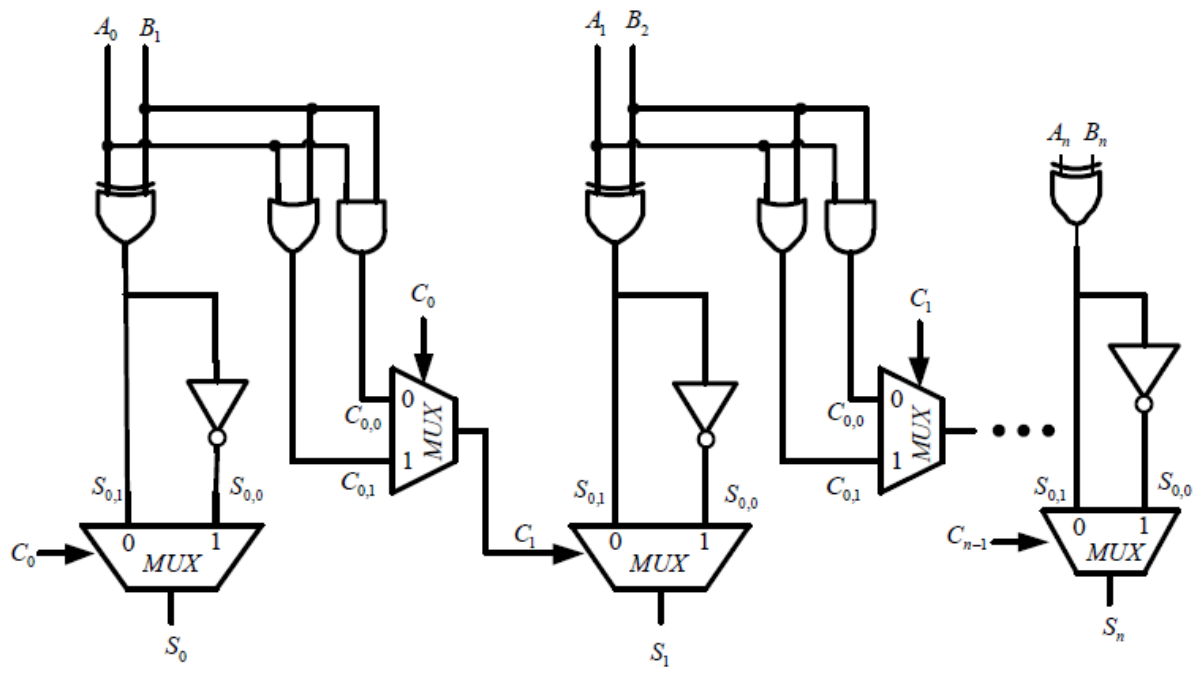


Fig 3.5 Internal Structure of CSLA using Common Boolean Logic [8]

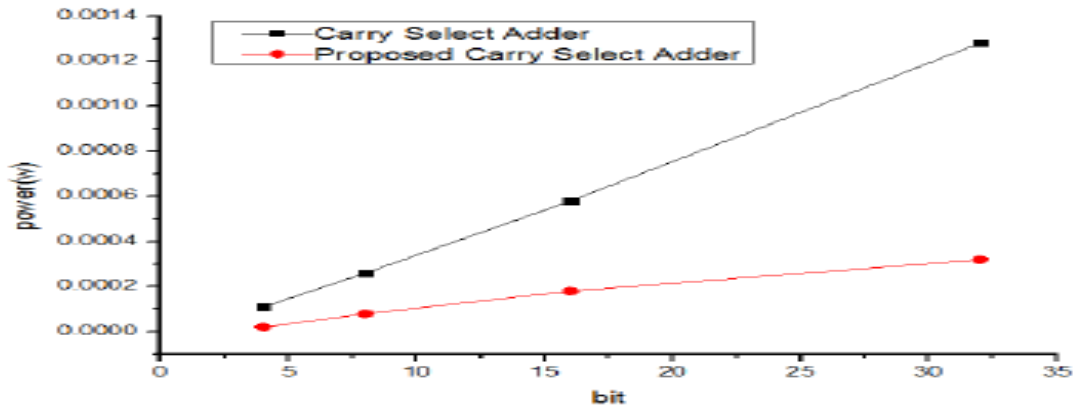


Fig 3.5 Power comparison between Sqrt CLSA using excess decoders and CLSA using Common Boolean logic. [8]

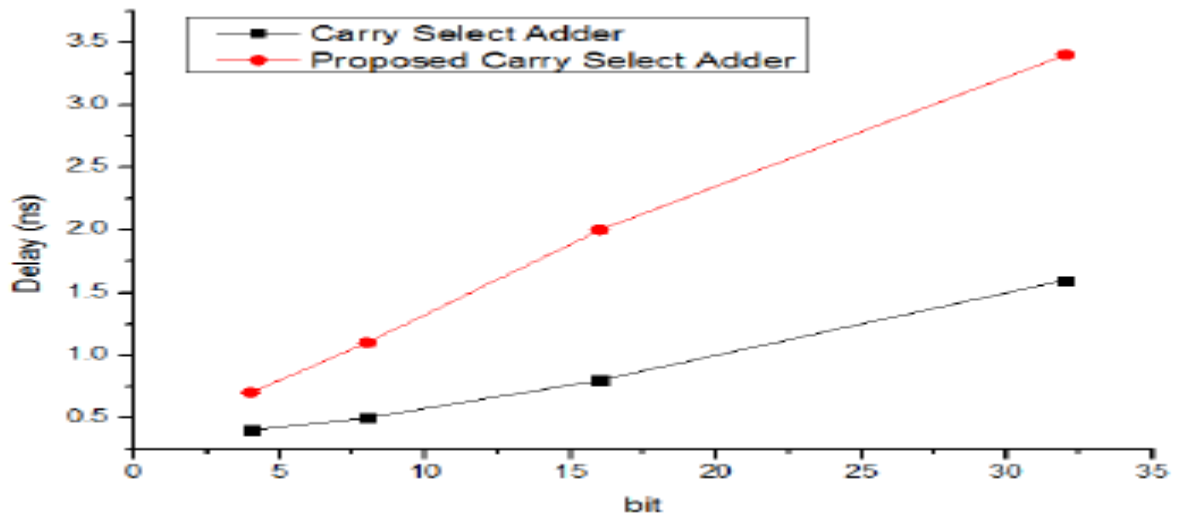


Fig 3.6 Delay comparison between Sqrt CLSA using excess decoders and CLSA using Common Boolean logic. [8]

Hence PDP of Sqrt CLSA using Common Boolean Logic is less than Sqrt CLSA using excess decoders. So we can optimise our modulo-m-adder by implementing more efficient adder used as subblock in it [8].

Chapter 4

Results and Conclusions

4.1 Results

Residues can be obtained for max 9-bit number as input in forward converter as moduli set for $n=3$ is $\{7, 8, 9\}$. Hence unique representation can be obtained for range $0 \leftrightarrow 503$.

Input Constraint

“number” is input constraint given in forward converter

1. r1 corresponds to residue of 2^n-1 adder
2. r2 corresponds to residue of 2^n adder
3. r3 corresponds to residue of $2^n + 1$ adder

“number 1” corresponds to output of reverse converter using SQRT CLSA with EXCESS 1 decoders.

“number 2” corresponds to output of reverse converter using SQRT CLSA with COMMON BOOLEAN LOGIC.

#0 number = 9'b110100101;

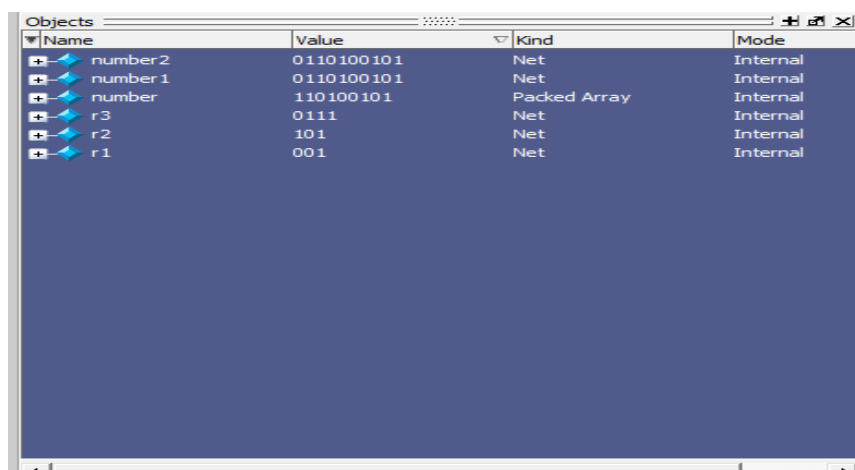
#100 number = 9'b110111101;

#100 number = 9'b110101101;

#100 number = 9'b110101100;

#100 number = 9'b110101111;

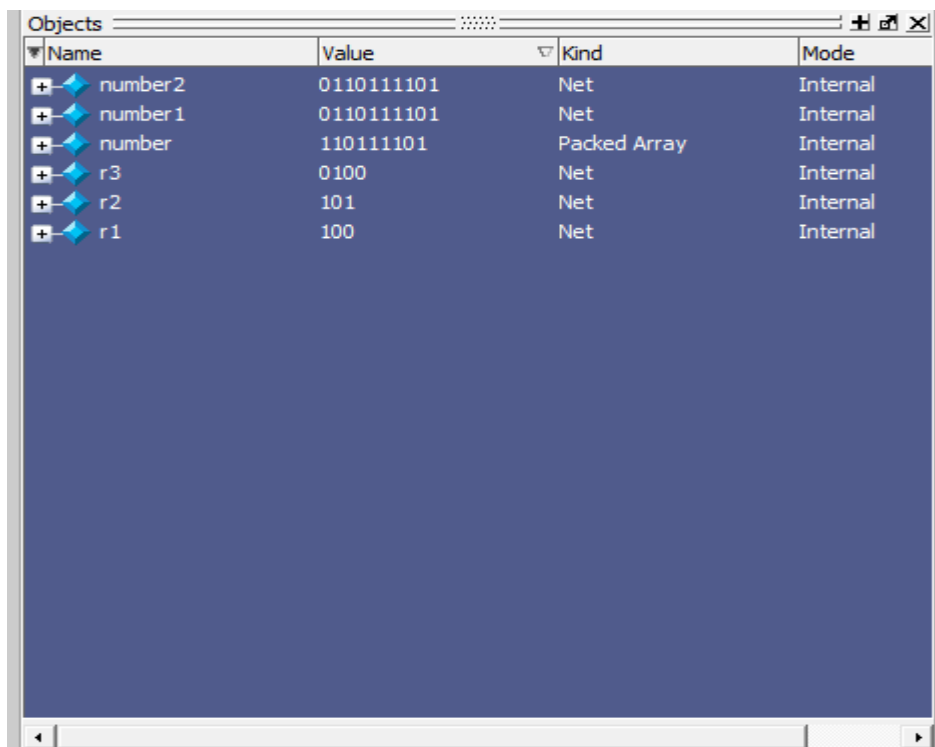
Case 1: number = 9'b110100101



Name	Value	Kind	Mode
number2	0110100101	Net	Internal
number1	0110100101	Net	Internal
number	110100101	Packed Array	Internal
r3	0111	Net	Internal
r2	101	Net	Internal
r1	001	Net	Internal

Fig 4.1 Object table for input 110100101

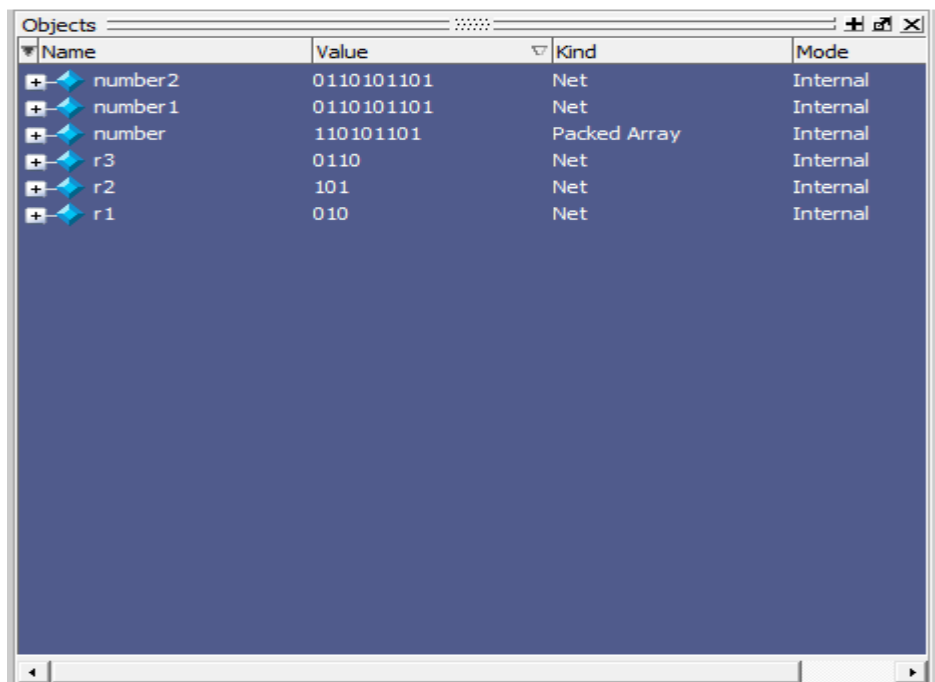
Case 2: number = 9'b110111101



Name	Value	Kind	Mode
number2	0110111101	Net	Internal
number1	0110111101	Net	Internal
number	110111101	Packed Array	Internal
r3	0100	Net	Internal
r2	101	Net	Internal
r1	100	Net	Internal

Fig 4.2 Object table for input 110111101

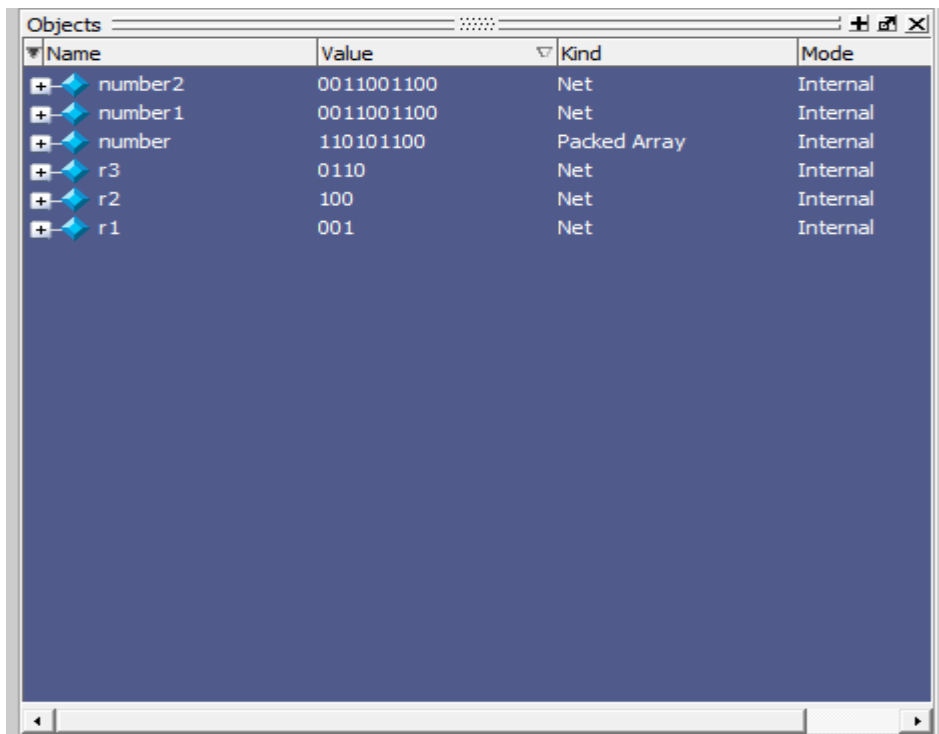
Case 3: number = 9'b110101101



Name	Value	Kind	Mode
number2	0110101101	Net	Internal
number1	0110101101	Net	Internal
number	110101101	Packed Array	Internal
r3	0110	Net	Internal
r2	101	Net	Internal
r1	010	Net	Internal

Fig 4.3 Object table for input 110101101

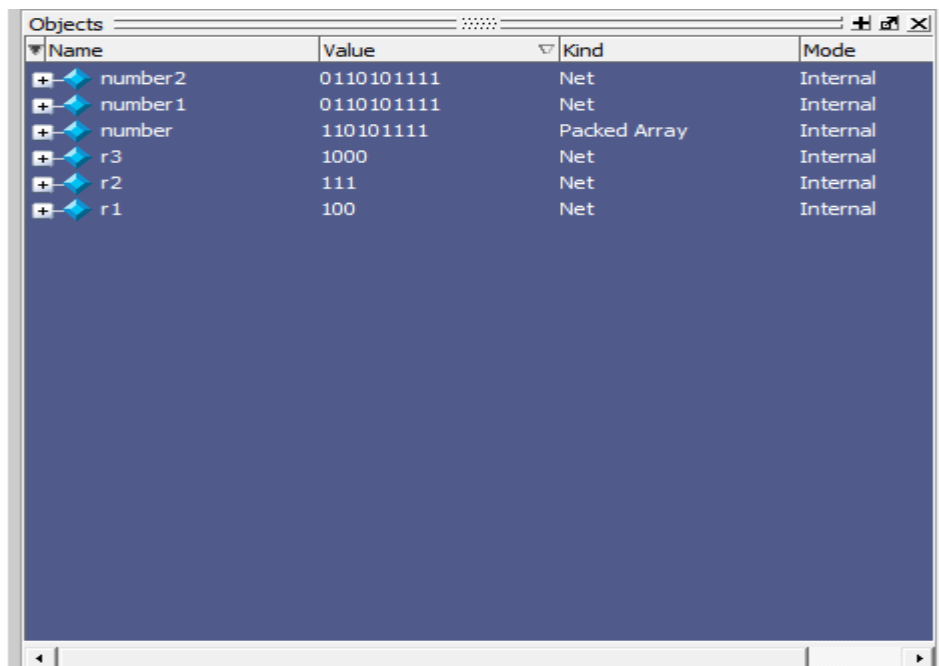
Case 4: number = 9'b110101100



Name	Value	Kind	Mode
number2	0011001100	Net	Internal
number1	0011001100	Net	Internal
number	110101100	Packed Array	Internal
r3	0110	Net	Internal
r2	100	Net	Internal
r1	001	Net	Internal

Fig 4.4 Object table for input 11001100

Case 5: number = 9'b110101111



Name	Value	Kind	Mode
number2	0110101111	Net	Internal
number1	0110101111	Net	Internal
number	110101111	Packed Array	Internal
r3	1000	Net	Internal
r2	111	Net	Internal
r1	100	Net	Internal

Fig 4.5 Object table for input 110101111

Combined output with waveforms :

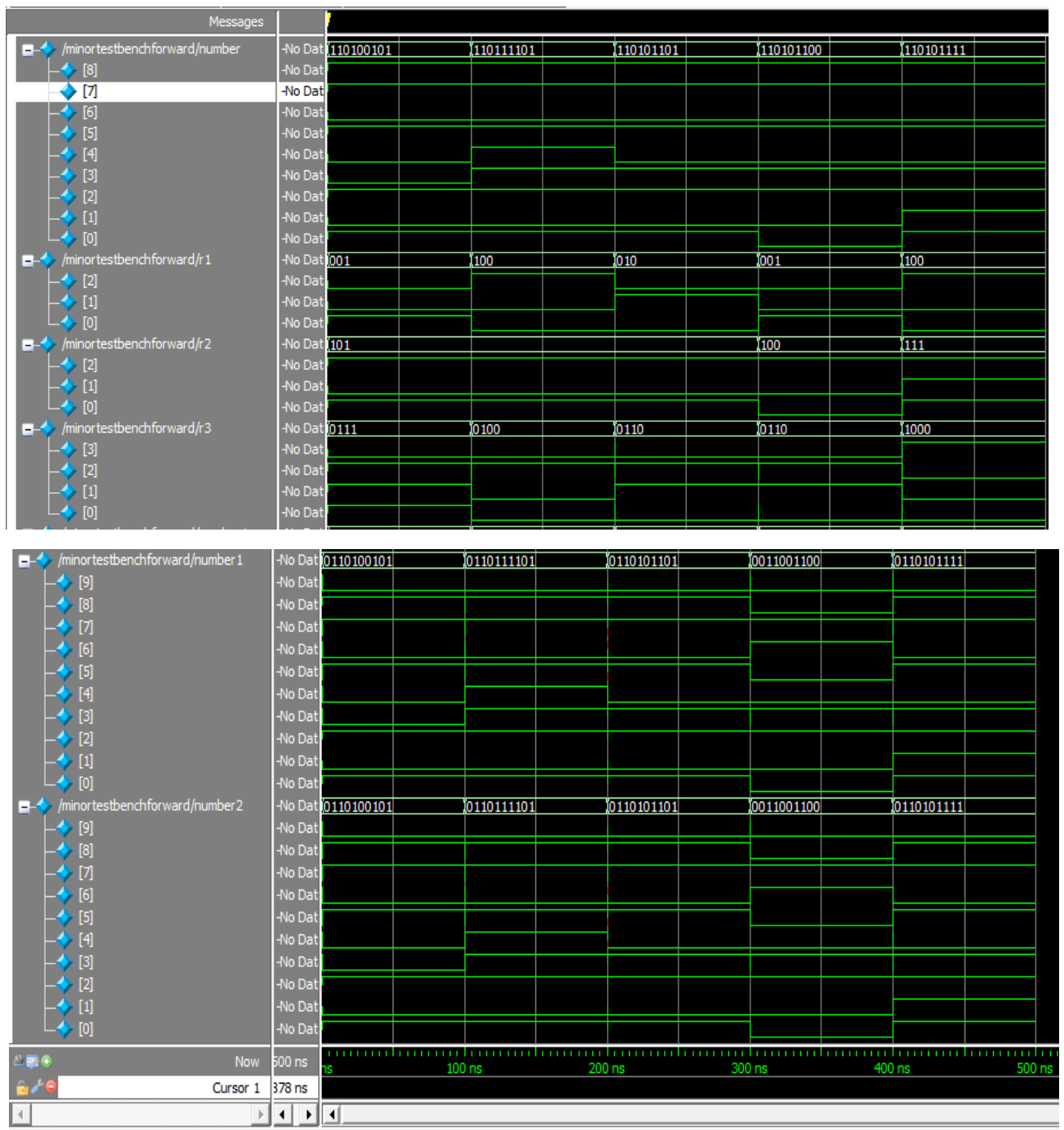


Fig 4.6 Complete waveforms for different input

4.2 Conclusion

1. From this project it can be concluded that max 9-bit number can be uniquely represented by special moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ for $n=3$. This Reduces the word length of data, reduces cost of ALUs and due their independency these residues will not affected by other residues if error occurs.
2. Different algorithms can be implemented and optimised using combinational logic for implementing modulo $2^n - 1$, modulo 2^n , modulo $2^n + 1$ adder.
3. Delays of multiplier in Reverse Converter can be replaced by LUT based ROMs to fasten the reverse converter.
4. Modulo-m-adder can be optimised by optimising adder used as subblock like shown in section 3.22.

REFERENCE

- [1] Samir Palnitkar, “Verilog HDL: A guide to Digital and Synthesis” Texts-1996.
- [2] Omar Abdelfattah, “Data Conversion In Residue Number System”, Canada McGill University january 2011.
- [3] Amos Omondi and Benjamin Premkumar, “Residue Number System: Theory and Implementation” Texts Vol-2 2007.
- [4] Jun Chen, “Parallel-Prefix Structures for Binary and Modulo Adders “ $\{2^n - 1, 2^n, 2^n + 1\}$ ” Texts 2008
- [5] G.C. Cardarilli and R.Lojacono, “RNS to Binary Conversion for Efficient VLSI Implementation”, IEEE transactions on circuits and systems—I: fundamental theory and applications, vol. 45, no. 6, June 1998.
- [6] Reto Zimmermann “Computer Arithmetic: Principles, Architectures and VLSI Design” lecture notes 16 March 1999.
- [7] Prof.Vojin G. Oklobdzija “High-Speed VLSI Arithmetic Units: Adders and Multipliers” Fall 1999.
- [8] I-Chyn Wey, Cheng-Chen Ho, Yi-Sheng Lin, and Chien-Chang Peng, “An Area efficient Carry Select Adder Design by Sharing Common Boolean logic” IMECS Vol-II March 14-16, 2016