

*A Fast VLSI Systolic Array
for Large Modulus Residue
Addition*

*S. Bandyopadhyay, G. A. Jullien, A.
Sengupta*

U N I V E R S I T Y O F

WINDSOR

V L S I R e s e a r c h G r o u p

A Fast VLSI Systolic Array For Large Modulus Residue Addition

S. Bandyopadhyay*, G. A. Jullien* and A. Sengupta**

*VLSI Research Group, University of Windsor

Windsor, Ontario, Canada N9B 3P4

**University of South Carolina, Columbia, SC 29208

Abstract:

The Residue number system (RNS) is inherently suited to high speed computations using custom tailored VLSI systems. In this paper, an algorithm for residue addition, based on a novel, 'non unique' number representation scheme, is implemented by a systolic array embedded in a VLSI chip. The pipelined cells are implemented, using a true single phase clock dynamic circuit structure, with computer synthesized minimized trees (switching trees). The array may be easily programmed by the user to accept any arbitrary modulus. Important applications of this array are in residue decoding and fault tolerant computation requiring the use of the Chinese Remainder Theorem where the modulus for addition is relatively large.

Keywords:

Residue number systems, Chinese remainder theorem (CRT), number theoretic transforms, residue adder, systolic arrays, dynamic logic.

I. Introduction

Recent advances in computer architecture and VLSI technology have brought about a resurgence of interest in RNS based digital systems [1][2]. In a residue number system, using a set of moduli $\{m_0, m_1, \dots, m_{N-1}\}$, a number X is represented by a set of residues $\{x_0, x_1, \dots, x_{N-1}\}$ where $x_i = X \bmod m_i$. If $X < M$ and $M = \prod_{i=0}^{N-1} m_i$, the set of residues is unique for any X , provided the set of moduli contains only relatively prime moduli. In RNS arithmetic, addition, subtraction and multiplication are inherently "fast" operations [3] due to the carry free property. Decoding a set, $\{x_0, x_1, \dots, x_{N-1}\}$, of residues to obtain any number $X, 0 \leq X < M$ may be carried out using the Chinese Remainder Theorem or Mixed Radix Conversion [4-10].

In this paper, our objective is to build a high throughput system for computing the sum (mod M) of a number of residues, where the ring modulus is large and arbitrary. There is no major problem in building area/time efficient VLSI arrays for residue addition if the value of the modulus is small [1][10][11]. There are, however, applications such as scaling, residue decoding, error detection and correction [1] which involve, in some way, an adder for a large modulus (typically 30-40 bits). In conventional residue addition, a correction stage is involved whenever the sum exceeds the value of the modulus. This correction stage, a checking procedure, which determines whether or not the sum exceeds M , and subsequent addition of the correction, are all time consuming and hardware intensive. The situation is exacerbated for large values of M with awkward structures (many non-zero bits in a binary representation). Such adders can be considered expensive both in terms of critical path and silicon area.

A typical example of an operation involving a large modulus, M , is a conversion operation using the Chinese remainder theorem (CRT). A straight forward implementation of the CRT involves the use of a large modulus multioperand adder. To avoid the problem of designing a

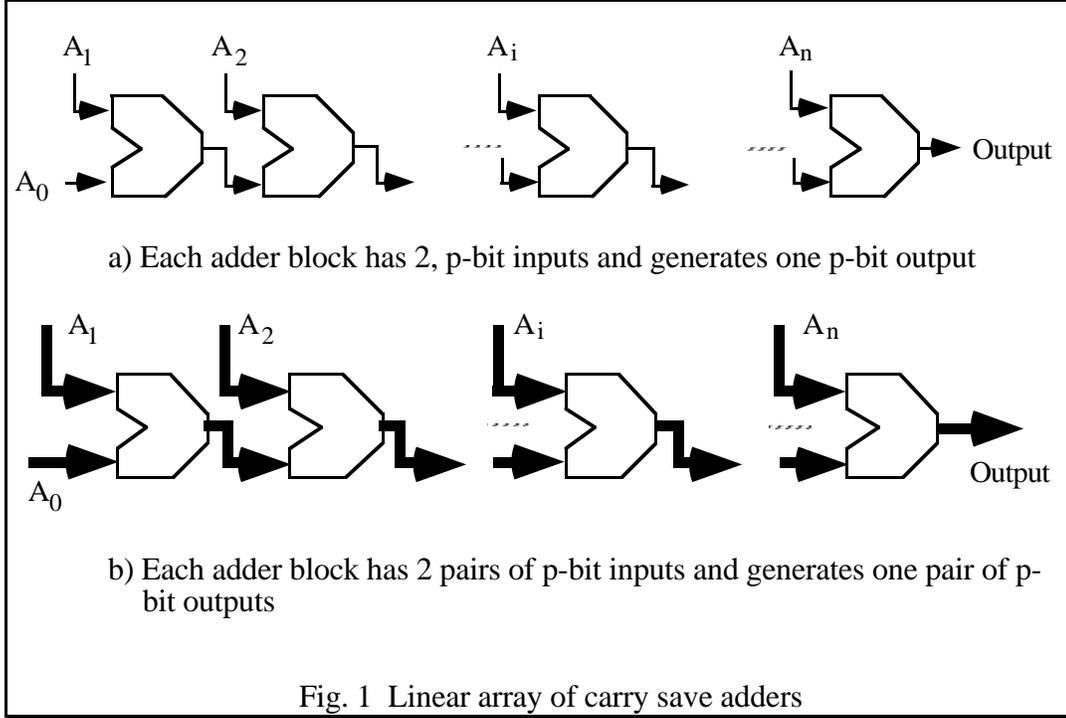
large modulus adder, investigators have resorted to the use of specially selected sets of moduli [7] and/or custom tailored hardware to take advantage of properties of the CRT [8][9].

The approach, presented in this paper, relies on constructing an adder where the size of the modulus is independent of the size of the primitive addition cell and number of pipeline stages required; also the throughput rate is maintained over the entire computational system and the adder structure is very regular. To meet these requirements, we have designed a systolic array for residue addition where we use carry save addition [12] and a special non-unique RNS representation[13]. The throughput of the array is determined by the speed of a single full adder, and a remarkable feature of the array is that the number of stages of addition, to add two residues, is constant (5) and thus independent of the size of the modulus. This property does not diminish the speed advantages of carry save addition. For large, arbitrary modulus residue addition this scheme produces high throughput pipelined designs with low latency.

To explain our approach we review, in section II, the problem of designing a pipelined multioperand adder to add N binary numbers A_0, A_1, \dots, A_{N-1} . The number representation scheme in section III is taken from [13] and is included here to make the paper self contained. We discuss the residue adder in section IV. This is an adaptation of the pipelined scheme outlined in section II to take care of residue operations. In section V, we present a VLSI implementation of the CRT, using this adder implemented in a true single phase clocked dynamic logic, and compare it to other approaches for implementing the CRT. We show that our approach requires hardware comparable to that needed in decoding circuits which exploit special properties of the CRT and/or use special moduli sets. Thus, in our scheme, there is no restriction on the choice of moduli and no need to develop special purpose hardware to circumvent the problems of large, arbitrary, modulus additions.

II. A Pipelined Multioperand Binary Adder

We will briefly review the conventional problem of adding N binary represented numbers A_0, A_1, \dots, A_{n-1} , using carry save adders to maximize throughput. To simplify the situation, let $A_i < 2^p$ and $\sum A_i < 2^p$. Two schemes are shown in figures 1a) and 1b).



In Fig. 1a), in the i th block, we add the i th number to the partial sum. The result, a p -bit number, is generated after a latency period of p cycles, where each block involves $p-1$ stages of full adders. The number of adders in the first stage is $p-1$ and this number decreases by 1 for each successive stage.

In Fig. 1b), we represent each number by a pair of p -bit numbers so that A_i is represented by the pair $(A1_i, A2_i)$ of p -bit numbers. Since we use carry save techniques $(A1_i, A2_i)$ represents sum and carry bits. Our objective is to compute a result which is also a pair of p -bit numbers. In this case, the operation in the i th block has two stages and involves exactly 2 rows of adders. The first stage uses carry save addition to add $A1_i$ to the partial sum generated by the $(i-1)$ th block. This produces a result in the form of a pair of p -bit numbers. The second stage adds $A2_i$ to the pair generated in the first stage, also producing a pair of p -bit

numbers. The scheme in Fig 1a requires $p(p-1)/2$ adders for each of the n adder blocks whereas the scheme of Fig 1b requires $2p$ adders. For $p > 5$, the scheme of Fig 1b requires less hardware than that of Fig 1a. Our algorithm is a modification of Fig 1b) where we perform residue addition instead of binary addition.

We mention, in passing, that the above explanation uses a linear array of carry save adders. Other structures, such as a tree of adders, are possible, and should be used when n is large.

III. A Scheme For Residue Number Representation

To represent the i th residue, x_i , of a number X (i.e. $x_i = X \bmod m_i$), the conventional method is to use p bits where $m_i \leq 2^p$. Thus $x_i = \sum_{j=0}^{p-1} 2^j a_i^{[j]}$ where $a_i^{[j]}$ is the j th bit of the i th residue.

Where it is obvious by context, we will drop the subscript i . Since $0 \leq x < m$, the only valid bit configurations are those whose binary weighted values are less than m . In this conventional representation, if the result $C = A \circ B > m$, $\circ \in \{+, -, \bullet\}$, a correction of $2^p - m$ is required. An additional subtraction operation is used to determine the magnitude of C and the correct, reduced, result selected.

The representation we will use in this paper, assumes that all p -bit configurations, $2^{p-1} < m \leq 2^p$, are valid. If N is the p bit binary weighted value, representing a residue x , then the mapping back to the conventional representation is given by:

$$x = \begin{cases} N & N < m \\ N - m & m \leq N \end{cases} \quad (1)$$

Thus residues in the range $0 \leq x < 2^p - m$, will have two possible representations $\{x, x + m\}$. The advantage of this non unique representation scheme is that corrections are applied only when there is a carry out of the MSB, and this operation can be effectively pipelined [13]. We note that this representation has also been used in the later work of Elleithy, et. al. [14].

The non unique representation may be immediately extended to allow operands to be pairs of p bit numbers for carry save operations. Thus the pair $(N1, N2)$ represents a number $x \bmod m$ whenever $x = N1 \oplus_m N2$, where modulo m addition is represented by the operator, \oplus_m . Clearly, depending on m and p , each value of x , $0 \leq x < m$ may be represented by many pairs $(N1, N2)$ where $0 \leq N1, N2 < 2^p$.

III.1 Example 1 :

Using binary notation, if $m = 101$, the residue 0 may be represented by $(110, 100)$, $(011, 010)$ or $(000, 000)$. However, if we add the first number (110) to the second number (100) we generate a carry out of the MSB and we need to correct the result. In our algorithm corrections are applied until there is no carry out of the MSB.

IV. Design of a Fast Residue Adder

Our objective is to compute the sum of two residues, based on the approach in Fig. 1b), where each of the two operands is represented by a pair of p -bit numbers, using the non unique representation discussed in section III. The result of adding these two operands, modulo m , will also be a pair of p bit numbers. In example 1, it is important to note that the pair $(110, 100)$ represents $(6+4) \bmod 5 = 10 \bmod 5 = 0$. In terms of our representation $(110, 100)$ is a perfectly valid representation of 0.

We first briefly discuss the algorithm, omitting details, in order to establish our technique. We then present the algorithm in detail.

IV.1 The Algorithm

Let the first residue be represented by the pair of numbers $(R1, R2)$ and the second by the pair $(A1, A2)$ where $R1$, $R2$, $A1$ and $A2$ are all p -bit numbers. The carry save adder requires 5

stages to ensure that no carry is output at the MSB. These stages are described below:

Stage 1) We add A_1 to the pair (R_1, R_2) generating a $(sum, carry)$ pair. There may be a carry, CP_1 , out of the MSB. We do not, however, check for carry overflow at this stage.

Stage 2) We add A_2 to the pair of sum bits and carry bits generated by stage 1. There may be a carry, CP_2 , out of the MSB. We do not apply any correction is applied at this stage .

Stage 3) Here we have to consider three separate scenarios, based on the conditions of the carry bits CP_1 and CP_2 . A carry, CP_3 , may be generated.

$(CP_1 = CP_2 = 0)$ The output of stage 2 is a valid representation.

$(CP_1 + CP_2 = 1)$ We add a correction of $2^p - m$.

$(CP_1 = CP_2 = 1)$ We add a correction of $2(2^p - m)$.

Stage 4) If $CP_3 = 0$ the representation is valid. Otherwise we add a correction of $2^p - m$. A carry, CP_4 , may be generated.

Stage 5) If $CP_4 = 0$ the representation is valid. Otherwise we add a correction of $2^p - m$. No carry will be generated and the algorithm terminates.

IV.2 Details of the Algorithm

To distinguish between the partial sums generated by the different stages, the following notation will be used. The inputs to the i th stage of operations include a pair of p bit numbers $(R_{1_{i-1}}, R_{2_{i-1}})$. The outputs are a pair of p bit numbers (R_{1_i}, R_{2_i}) and a bit, CP_i , representing the carry out of the MSB. If there is no carry, the stage and the corresponding output will be termed *carry free*.

The algorithm is detailed in Table 1 and generates a carry free pair (R_{1_5}, R_{2_5}) at the output. The

inputs to stage 1 are the pairs $(R1_0, R2_0)$ and $(A1, A2)$. The operation in the i th stage is $[(R1_{i-1}, R2_{i-1}) + X]$.

Stage i	Operation		
$i = 1, 2$	$(CPi, R1_i, R2_i) = [(R1_{i-1}, R2_{i-1}) + Ai]$		
$i = 3$	$CP1$	$CP2$	Operation
	0	0	$CP3 = 0$ $R1_3 = R1_2$ $R2_3 = R2_2$
	0	1	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + (2^p - m)]$
	1	0	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + (2^p - m)]$
	1	1	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + 2 \cdot (2^p - m)]$
$i = 4, 5$	$CP(i-1)$		Operation
	0		$CPi = 0$ $R1_i = R1_{i-1}$ $R2_i = R2_{i-1}$
	1		$(CPi, R1_i, R2_i) = [(R1_{i-1}, R2_{i-1}) + (2^p - m)]$

Table 1 The 5-stage algorithm for Carry-Save RNS addition

It is important to note that in the correction stages 3, 4 and 5, if there is no carry in the preceding stage, no addition operation is performed on the sum or carry bits. Even if a particular pair of sum and carry bits is valid in terms of our non unique representation, if we perform an addition of the sum and carry bits, the result may include a carry out of the MSB, requiring correction. For example, the pair (110,100) is a valid residue representation of 0 mod 5. If we add 110 to 100, we get (010, 000) with a carry out of the MSB. This carry immediately requires a correction stage, which we wish to avoid.

It is a crucial property of our algorithm that there must be no carry out of the MSB, in stage 5, in order to guarantee a valid representation. This is formally established in theorem I below.

We will define an operator $MSB(x)$ to denote $x^{[p]}$ (the MSB of result x).

Theorem 1

A maximum of three stages of correction are required to ensure that $CP5 = 0$.

Proof

If $CP1 = CP2 = 0$, then $CP3 = CP4 = CP5 = 0$; otherwise, the correction is $(2^p - m)$ ($2(2^p - m)$) depending on whether one (both) of $CP1, CP2$ is (are) 1. Since $2^{p-1} < m \leq 2^p$, $MSB(2^p - m) = 0$ but $MSB(2(2^p - m)) \in \{0,1\}$. The worst case carry generation occurs when $CP1 = CP2 = MSB(R1_2) = MSB(R2_2) = MSB(2(2^p - m)) = 1$. In this case, $CP3 = MSB(R1_3) = 1$, and $MSB(R2_3) \in \{0,1\}$ and, in stage 4, a correction of $(2^p - m)$ is added to the pair $(R1_3, R2_3)$, where $MSB(2^p - 1) = 0$. If $CP4 = 1$ clearly $MSB(R1_4)$ must be 0, so that when a correction of $(2^p - m)$ is added in stage 5 to the pair $(R1_4, R2_4)$, it is guaranteed that $CP5 = 0$. Other cases are similar. □

IV.3 Example 2

Add $(101,101)$ to $(110,011)$ using modulus 101. $(A1, A2) = (101,101)$ and $(R1, R2) = (110,011)$. The result is the pair $(101,100)$ which is equivalent to 100, as shown in Table 2.

Stage	Partial Sums and the carry
1	(000,110, 1)
2	(011,000, 1)
3	(101,100, 0)
4	(101,100, 0)
5	(101,100, 0)

Table 2

IV.4 Comments

Conventional residue addition requires a correction to be added whenever the sum of two residues exceeds the modulus. Consider a situation where we use a large modulus, M . The process of checking for overflow from the sum of two residues has to be done using a bit by bit comparison starting from the MSB. The time required for this magnitude comparison is $\tau(\log_2 M)$, assuming that the sum of two residues is a single binary number with $\lceil \log_2 M \rceil$ bits, and τ is the addition logic propagation

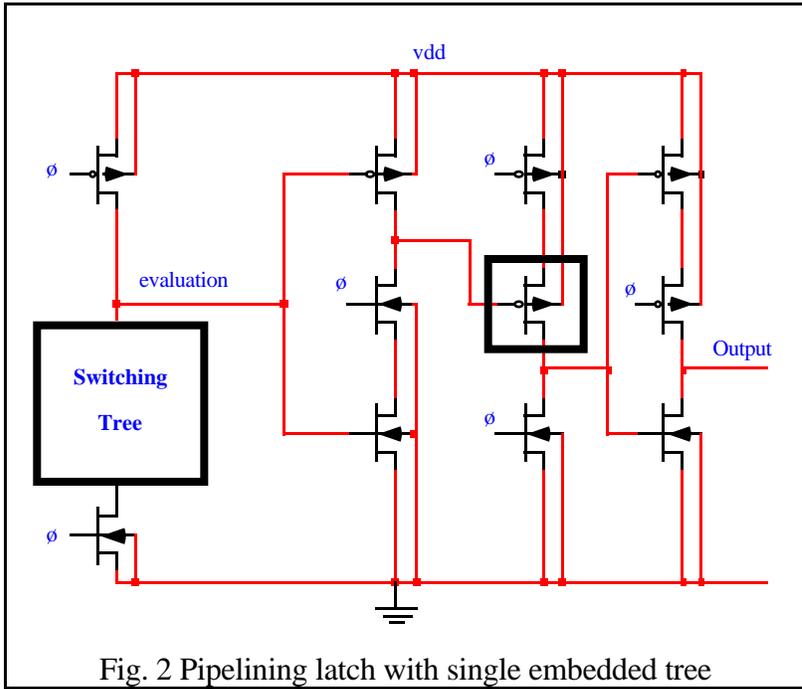
delay. If we wish to design a fast residue adder it is tempting to use carry save adders since the advantages of carry save addition for fast arithmetic [12], using systolic arrays, are well known. But in carry save arithmetic, the sum is in the form of carry bits and sum bits. Therefore we need an additional $\tau(\log_2 M)$ time just to add the carry bits and sum bits to obtain a single number with $\lceil \log_2 M \rceil$ bits. In other words we have lost the advantage of carry save arithmetic and in a conventional residue addition scheme, the time required to generate the sum of two residues is $O(\log_2 M)$. Since this time determines the clock rate, clearly large modulus residue addition is time consuming. As mentioned in the introduction, applications involving large moduli attempt to circumvent this problem by using specially selected moduli [1]. Our approach uses a constant number of stages irrespective of the size of the modulus, and for applications where speed is a critical issue, our algorithm is very appropriate. A typical application is to use this technique for the final stage of a fast signal processing application for decoding and/or automatic error correction.

V. CMOS VLSI Implementation

The algorithm described above may be directly implemented by a systolic array. Realizing the cells for a conditional adder by conventional logic gates is not very efficient since each stage implements a slightly different operation, and logic gate decompositions may not yield the optimum *Area.Period* cost function we require. The approach that we have chosen is to use *switching tree cells* embedded in a dynamic pipelined block, and synthesized with *Woodchuck* [15]. The dynamic circuitry uses a *true single phase clock* structure [16].

The basic concept behind switching tree cells is to implement the switching function as a look-up table, but to construct the table as a minimized binary tree switching circuit. The minimization is based on the electrical characteristics required of the switching circuit, and does not involve either the usual Boolean algebra minimization or the concept of logic gate primitives [15].

V.1 Pipelined Switching Trees

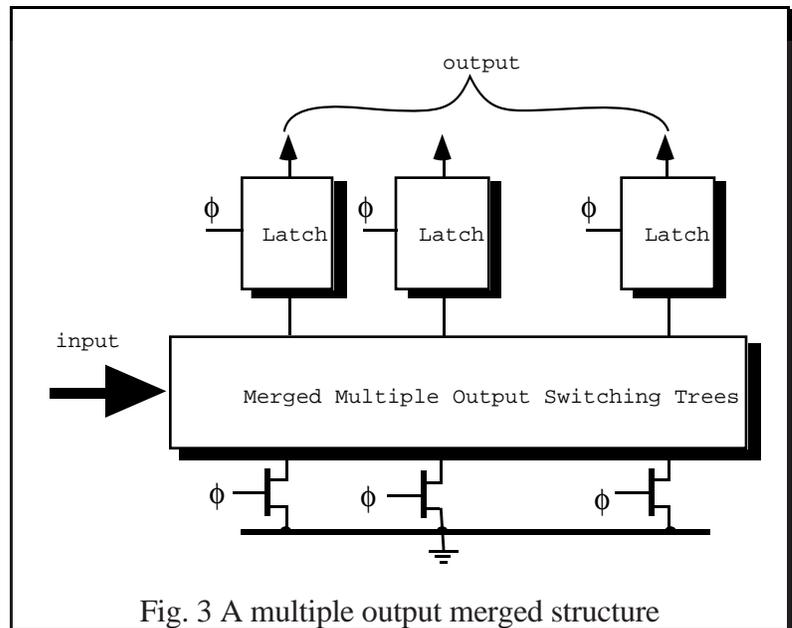


We embed a complex NFET logic block (switching tree) in a TSPC master/slave latch, as shown in Fig. 2. Note that the p-channel logic block (highlighted) is restricted to a single PFET (inverter). Our approach is to build the logic for each stage entirely within the NFET block; this provides the most area efficient

implementation of a given logic function. For a switching function with multiple output bits, we provide a separate latch structure for each bit, but attempt to merge the embedded trees over the complete set of output bits. This is shown in block diagram form in Fig. 3.

The tree is designed as an n -dimensional ROM (binary tree) where n is the number of input variables, as shown in Fig. 4.

Our notation represents transistors whose gates are driven by the true logic input as arcs, \backslash ; the other arc, $/$, represents transistors whose gates are driven by the



complement of the logic input. By removing selected arcs from the bottom of the tree, we can implement any arbitrary truth table.

Viewing the tree in this way allows tree height reduction using higher order decoders rather than the single inverter decoders required for the n -dimensional ROM. A full binary tree possesses interesting qualities as far as a series chain discharge block in dynamic logic is concerned. In the

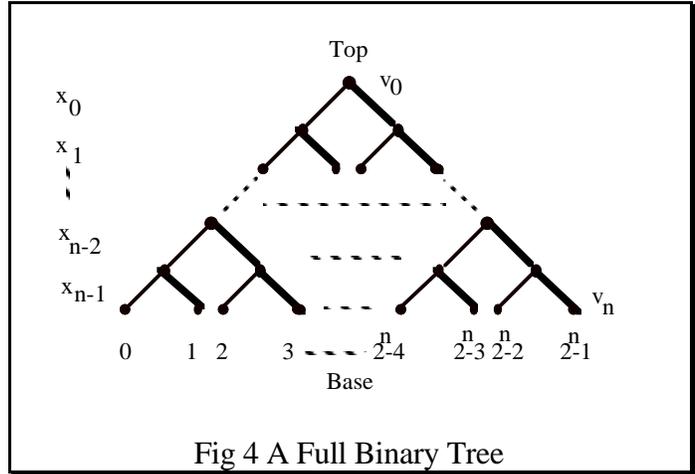


Fig 4 A Full Binary Tree

full tree we see that, for stable logic inputs, only a single series path connects the top node to one of the bottom nodes, and that the capacitance at every node in each of the possible series paths is 3 source/drain capacitances in parallel. The gate load at the i th level of the tree is $C_G 2^i$, where a single transistor gate load capacitance is C_G . This latter result is, perhaps, a disadvantage for the use of full binary trees as logic blocks, but the former results are advantageous. Some of these qualities are modified when the tree is minimized.

Our minimization technique is not based on Boolean algebraic concepts, as with most reported dynamic logic designs, but rather on the application of two graph reduction rules. We find this approach useful in that it allows a well established relationship between reduced tree structure and silicon layout that is essential for both hand custom layout and module generation approaches for complex multiple output trees.

Based on fabricated cells, we find that a 6-level switching circuit can be dynamically pipelined at rates exceeding 50MHz in a 3μ CMOS process [17]; a 4-level tree (the maximum obtained in our residue adder design) will pipeline at much higher rates.

V.2. Graph Based Reduction

In order to present the two simple rules used in the minimization procedure, the following definitions are given:

A tree represented by a graph can be denoted as, $G = \{X, V\}$, where X is a set of edges (n-channel transistors) $\{x_{i,j}\}$, and V is the vertex set of nodes $\{v_{i,j}\}$. An edge $x_{i,j}$, consists of elements $(ct, v_{k,l})$, where i and k are tree levels, $j \in [0, 2^{i+1} - 1]$, $l \in [0, 2^k - 1]$, and connection type $ct \in \{T, F, W\}$. The inputs to the tree are $g_i \in \{0, 1\}$. If $g_i = 1$, then the path takes edge T if it is present. If $g_i = 0$, the path takes edge F if it is present. W represents an arc which is a wire, or link, connection, and is only present following the successful application of a reduction rule.

A path, $P_{(i,j),(k,l)}$, is the connection from node $v_{i,j}$ to node $v_{k,l}$, constructed by edges. A full path connects node $v_{0,0}$ to node $v_{n,l}$, where n is the height of the tree. A switching tree is the reduction of a unique set of full paths that describe a logical function. A tree is characterized by two sets of full paths, a true set in which an edge T or F is present at the n th level, and a complement set in which an edge T or F is removed at the n th level. A truth table is mapped onto a full tree by removing a sub-set of edges $\in \{x_{n,j}\}, j \in [0, 2^{n+1} - 1]$, from a full tree based on the set of zeros in the truth table.

The following two rules are used in the graph reduction technique; we omit proofs for brevity.

Rule 1: Merging of shared sub-trees

If paths from $v_{i,j}$ to $v_{n,l}$ and from $v_{i,k}$ to $v_{n,m}$, where $j, k \in [0, 2^i - 1]$, $l, m \in [0, 2^n - 1]$, contain an identical set of edges, starting at a node at level p , those nodes where the matching occurs in both sequences can be merged. Furthermore, if $k = j$, and $i - p = 1$, then the edges from node $v_{i,j}$ to nodes $v_{p,l}$ and $v_{p,m}$ can be replaced by a link edge.

Rule 2: Merging of Common Edges (Wires)

Consider a set of edge paths, X_1 , connecting a node $v_{i,j}$ with a node at level n , and a set of

edge paths, X_2 , also connecting the node $v_{i,j}$ with a node at level $n..$ Path X_1 follows the T edge from node $v_{i,j}$, and path X_2 follows the F , edge from node $v_{i,j}$. If X_2 covers X_1 , then the first edge in X_1 has $ct = W$.

Rule 1 provides for the greatest reduction in the number of nodes by merging common subtrees. Rule 2 replaces transistor links between nodes with wire links. When merging occurs, however, accidental paths (*sneak paths*) through the tree may be created which can produce false results. These sneak paths must be detected and either the tree reduction that caused them reversed, or duplicate paths introduced that effectively block the sneak paths.

Rule 2 provides an important reduction mechanism, however, when the truth table contains *don't care* states. These states are set to either a 1 or 0 to facilitate tree decomposition. Rule 2 sets these states to force one half of a subtree to be a subset of the other half so that the transistor link leading to the subset may be replaced by a wire link. States in the covering half of the subtree that match those in the subset will always be taken care of by the subset. Thus, their effect on the output is unimportant. This may allow the use of Rule 1 in the lower portions of the switching tree (below the common edge).

The ordering of variables can have some effect on the minimization properties of the graph based reduction technique. This effect is somewhat mitigated as we merge several trees (multiple output bits) using a common input variable order, since the optimum ordering will be different for each tree. We use a simple cyclic input variable shift to examine a small subset of the possible search space. We will see, in the next section, that often the footprint of a synthesized cell is bounded by the number of input inverters and output latches, rather than the area of the minimized merged trees. Thus the requirements for finding a global minimization solution (an NP complete problem) are mitigated.

V.3 Synthesizing Pipelined Switching Trees

We have developed a generator to produce switching tree modules. The module floor plan is shown in Fig. 5.

The transistor block consists of horizontal transistor positions in a grid (only the required transistors are constructed) with strips of polysilicon and metal2 running vertically, as shown in Fig. 6 for the Stage 3 merged tree. The poly/M2 strips both connect the p and n-channel inverter strips forming both distributed static inverters, and gate signals for the transistors in the merged trees. Use of metal2 provides low resistance connectivity between the two halves of each inverter, and also can provide low resistance connections to the transistor gates via poly/M2 shorting primitives.

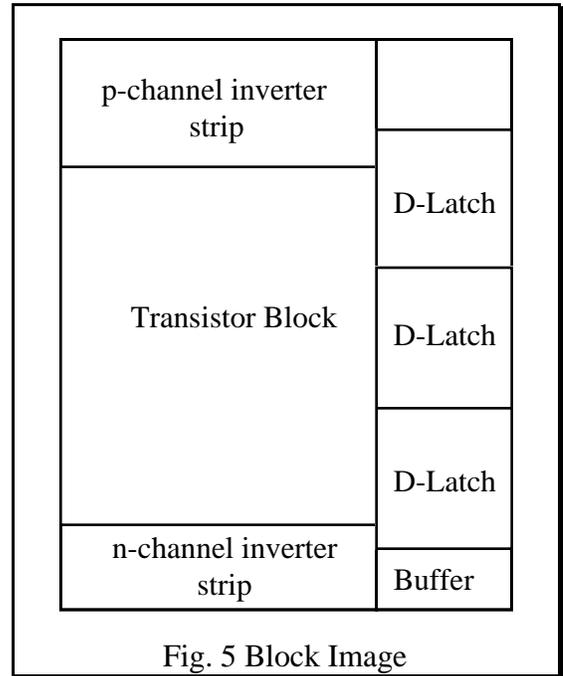


Fig. 5 Block Image

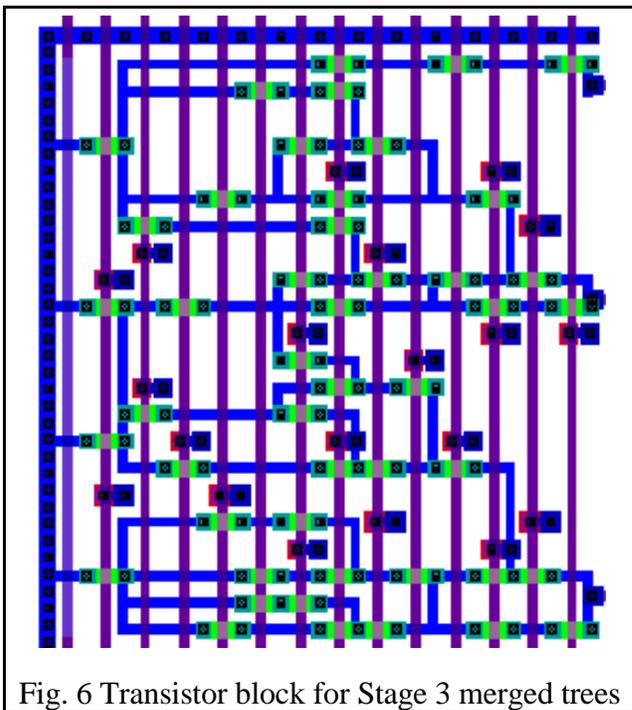


Fig. 6 Transistor block for Stage 3 merged trees

The output of the trees are connected to a set of D-Latches, where the particular tree forms part of a corresponding latch. The inverter strip provides clock signals to the ground switch(es) on each tree, thus buffering the clock. The buffer block provides clock buffering to the D-latches. The input clock load is therefore 2 inverters. The transistor block is formed by mapping the trees directly to a transistor or wire primitive. Table 3 shows the merged trees for the stage 3 switching

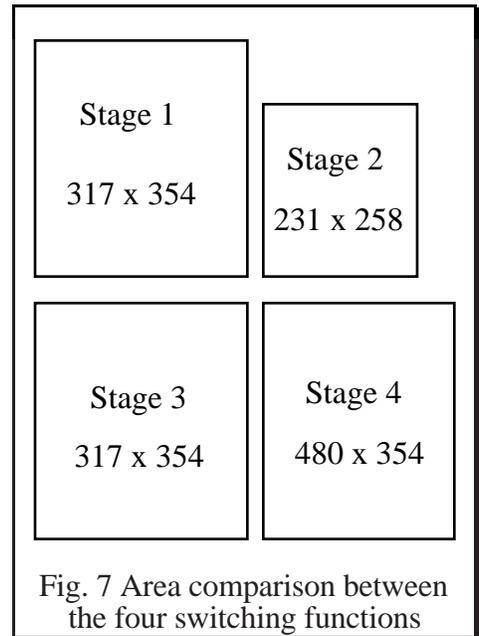
function, where each element in the table is mapped to a layout primitive. The rows alternate between cross connections and vertical transistors (the tree is perpendicular to the final block orientation shown in Fig. 6).

W								W										W	W		
F						t		T	F	f							t		T		F
					W	W	W								W	W	W	W			
F				t	T			F	T	t				f	F			t	T		T
			W	W				W				W	W	W					W	W	
W			T		W		t	T	W		f	F			T		t		F		T
	W	W				W	W				W		W	W			W	W		W	
T	T		F	t	T	T		F	T	f	F	T	F	t	T		F	f	T	F	T
			W	W				W	W	W		W									
W	F				T	W			W				W		W	t	T		F	F	T
W	W				W	F	f		T			f	F	t	T		W		W	W	W
W	W	W	W	W	W				W	W	W	W	W	W			W	W	W	W	
			F					f	F					F		f			F		

Table 3 Merged trees for the stage 3 switching function

Alternate rows, starting at the top, are for cross Metal1 connections (*W* operator). The other rows contain transistor and vertical wiring information.

The transistors are represented as *T* (gate is connected to true input signal) or *F* for a complement input signal; the wire primitives are labeled as *W*. In order to place the transistors as close together as possible, the true/complement ordering is reversed every other row. The *t* and *f* operators are Poly/M2 shorting primitives that are used to bring the inverter signal on the low resistance M2 conductor down to the Poly gate. The *t* short lines up with a *T* transistor on the particular row, and an *f* short lines up with an *F* transistor.



An area comparison between the four switching

functions, used to realize a general purpose mod M adder, is shown in Fig. 7, for a 3μ DLM CMOS process.

VI. Implementing The Chinese Remainder Theorem

The method for converting the residue representation $\{x_0, x_1, \dots, x_{N-1}\}$ of a number X into a natural integer is based on the Chinese remainder theorem [3]:

$$X = \sum_{i=0}^{N-1} \left\{ \hat{m}_i \otimes_M \left[(\hat{m}_i)_i^{-1} \right] \otimes_M x_i \right\} \quad (2)$$

\otimes_M represents multiplication modulo M and \sum_M is the summation operator modulo M . $\hat{m}_i = \frac{M}{m_i}$, $X \in R(M)$, $x_i \in R(m_i)$ and $(\bullet)_i^{-1}$ is the multiplicative inverse operator, mod m_i . Since $(\hat{m}_i \otimes_M m_i) = 1$ the inverse, $\left[(\hat{m}_i)_i^{-1} \right]$, exists. We can simplify eqn. (2) to:

$$X = \sum_{i=0}^{N-1} \left\{ W_i \otimes_M x_i \right\} \quad (3)$$

The term W_i is called the unit metric vector [10].

VI.1 A 29-Bit Modulus Example

Let us consider a typical situation where we wish the dynamic range to be about 32 bits. Because of the flexibility of an arbitrary CRT modulus we elect to choose four 8-bit primes; this will allow the use of index addition, rather multiplication, for the main computation requirements. The maximum modulus is $251 \times 241 \times 239 \times 233 = 2^{31.649}$, which would normally be considered a difficult modulus for a direct CRT implementation.

A straight forward implementation of the CRT, using our scheme, is shown in fig 8. We use look-up techniques to generate $W_i \otimes_M x_i$ for the i th tree, $0 \leq i < 4$. The look-up tables can use modifications of the switching tree implementation, where we exchange more complex

decoders for a reduced tree height. We have 4 outputs, each corresponding to a modulus, where each of them is a 32 bit number. Since our adder adds 2 pairs of operands, we group the 4 numbers into 2 pairs of 32 bit numbers: $(x_0, x_1), (x_2, x_3)$. A pair of 32 bit numbers is represented by a bold line in fig 8. We need a single residue adder to add these 4 residues.

As discussed above, the residue adder has 5 rows of cells where each row has 32 cells. Therefore we need a total of $5 \times 32 = 160$ cells; based on the cell area in Fig. 7 and allowing 10% overhead for routing (quite conservative for the regular design) we have a final silicon area of about 22mm^2 for the 3μ process. The output of the adder is a pair of 32 bit numbers which have to be reduced to a single 29 bit number. Since this process

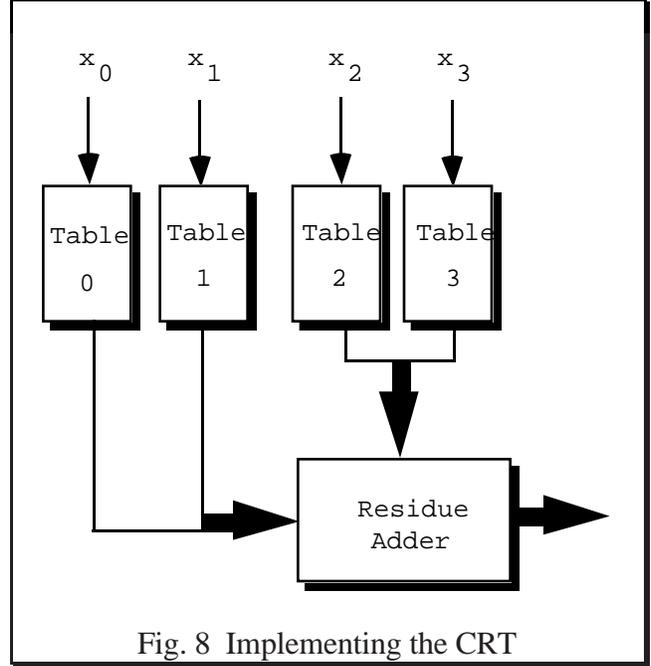


Fig. 8 Implementing the CRT

is straight forward and uses basically the same biasing techniques as used in [9], we omit this detail. We expect that a complete 32-bit CRT block will fit on less than 10mm^2 of silicon using a 1μ process.

VI.2 Comparisons with Recently Published Techniques

We are now in a position to compare our technique with recently published techniques for implementing the Chinese Remainder theorem. We will first briefly discuss each technique and then present a comparison table.

Vu's technique [9] represents a very economical architecture for realizing the CRT, providing that one of the moduli is a power of 2. This will not allow the type of index processing scheme we assumed in IV.1. If we use the same example of a 32 bit dynamic range (with one of the

prime moduli replaced by 128), we need 4 look up tables containing the (quotient, remainder) pair with respect to $\frac{M}{128}$ [9]. Essentially we have two outputs for every ROM - an eight bit output and a 24 bit output. We need one adder to add four 8-bit numbers and one adder to add four 24-bit numbers. To ensure that the speed of the circuit is comparable to ours, we have assumed that carry save adders are used throughout, with similar implementation techniques. In Vu's scheme, we also need another lookup table which we will ignore in our approximate comparison. The result is a pair of 32-bit numbers which is converted to a single 32 bit number using two 32-bit adders.

Zhang [18] has described a systolic tree structure for computing the CRT. This is a modification of Jenkin's approach [4] where the correction operation is combined with the next biased operation. Once again, we assume that carry-save adders are used throughout. The output of the adder is a single 32-bit number. The last stage of the network is a special cell which subtracts a bias of $2^p - M$ if needed. We have omitted this cell in our comparison so that all three methods essentially generate a pair of outputs.

We have not included Taylor's circuits [7][19] in our comparisons since his approach is based on a special 3 modulus system. The scheme proposed in [7] for large M (say 32 bits), involves the use of look up tables implemented using ROMs with a 12-bit address. Another recent paper [19] gives a scaling algorithm for CRT based on the 3-modulus system. We have excluded this paper since it addresses a slightly different problem; namely, efficient implementation of scaled CRT mapping using the moduli set $\{2^n - 1, 2^n \cdot 2^n + 1\}$.

Table 4 gives a comparison of the complexity of the three schemes for implementing the CRT in terms of the estimated silicon area needed for implementation. In this table we have considered three choices of moduli. The first set consists of 4 moduli, 8 bits or less, one of which is 128. The second set consists of 6 moduli, 6 bits or less including the modulus 64. The third consists of 8 moduli, 5 bits or less, one of which is 32. In all three cases, we

assume that the decoded value is a 32 bit number. It is interesting to note that the area needed by the new method is quite sensitive to the number of moduli. The area efficiency of the new approach, when the number of moduli is 4 or 6 is particularly interesting since our approach does not depend on the choice of moduli.

Method	4 moduli, each 8 bits or less	6 moduli, each 6 bits or less	8 moduli, each 5 bits or less
Vu	48.02 mm ²	42.45 mm ²	43.62 mm ²
Zhang	72.94	136.84	170.2
New method	19.96	39.92	59.88

Table 4 Silicon Area Comparison

VII Comments and Conclusions

Residue addition involves three operations: adding two residue using conventional arithmetic, checking if the result exceeds the modulus and, if the comparison in the previous step is successful, adding a correction. If the residues have p bits each, addition of the residues, magnitude comparison and addition of the correction, each requires $O(p)$ time. Carry save addition may be used to reduce the time required in the first and the third step of the process to $O(1)$. The problem is that of magnitude comparison, which requires either $O(p)$ time or $O(p)$ number of stages. The problem becomes particularly acute for large p .

In this paper, we have discussed two interesting concepts: a non unique representation for residues; and a fast VLSI implementation using the "switching tree" concept. We have shown that the scheme for representing residues may be used for developing algorithms for residue addition which can be readily mapped onto a systolic array. The important feature of this array is that it is inherently as fast as a standard carry save adder and thus can be pipelined at large bandwidths. There is a remarkable consistency in that each addition always takes 5 stages independent of the size of M ; in other words, we do not have a growth of addition stages as

the value of the modulus increases. Such large modulus adders are used in areas such as decoding using the Chinese Remainder theorem, error detection and correction, magnitude comparison and scaling. Traditionally, for fast implementations, such applications use specially selected sets of moduli and/or properties of the algorithm being implemented. Our approach here is to use a residue adder where the speed is determined by the speed of a single cell and the size of the modulus has no bearing on the number of stages of a systolic adder. We have shown that, in the case of the Chinese Remainder theorem, the approach often requires less the same hardware complexity as the best known algorithms, and offers no impediments associated with specially selected moduli. A standard systolic array can be used for any residue additions. The other important aspect of this paper is the concept of using switching trees for VLSI implementation. We have shown that the array, which requires 5 stages using 4 types of slightly different cells, may be efficiently implemented using minimized switching trees.

VIII References

1. Soderstrand, M.A., Jenkins, W.K., Jullien, G.A., and Taylor, F.J., ed. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. 1986, IEEE Press.
2. Taylor, F.J., 1984. "Residue Arithmetic: A Tutorial with Examples." *IEEE Computer Magazine*, **17**, pp. 50-62.
3. Szabo, N.S. and Tanaka, R.I., *Residue Arithmetic and its Applications to Computer Technology*. 1967, New York: McGraw-Hill.
4. Jenkins, W.K., 1973. "A Technique for the Efficient Generation of Projections for Error Correcting Residue Codes." *IEEE Trans. Comput.*, **C-22**, August, pp. 762-767.
5. Jenkins, W.K., 1978. "Techniques for Residue-to-Analog Conversion for Residue-Encoded Digital Filters." *IEEE Trans. Circuits Syst.*, **CAS-25**, July, pp. 555-562.
6. Baraniecka, A. and Jullien, G.A., 1978. "On Decoding Techniques for Residue Number

- System Realizations of Digital Signal Processing Hardware." IEEE Trans. Circuits Syst., **CAS-25**, November, pp. 935-936.
7. Taylor, F.J. and Ramnarayanan, A.S., 1981. "An Efficient Residue-to-Decimal Converter." IEEE Trans. Circuits Syst., **CAS-28**, December, pp. 1164-1169.
 8. Soderstrand, M.A., Vernia, C., and Chang, J.H., 1983. "An Improved Residue Number System Digital-to-Analog Converter." IEEE Trans. Circuits Syst., **CAS-30**, December, pp. 903-907.
 9. Vu, T.V., 1985. "Efficient Implementations of the Chinese Remainder Theorem for Sign Detection and Residue Decoding." IEEE Trans. Comput., **C-34**, July, pp. 646-651.
 10. Jullien, G.A., 1978. "Residue Number Scaling and Other Operations Using ROM Arrays." IEEE Trans. Comput., **C-27**, April, pp. 325-336.
 11. Bayoumi, M.A., Jullien, G.A., and Miller, W.C., 1987. "A Look Up Table Methodology for RNS Structures Used in DSP Applications." IEEE Trans. CAS, **CAS-34**, June, pp. 604-616.
 12. Hwang, K., *Computer arithmetic: Principles, Architecture and Design*. 1979, John Wiley.
 13. Bandyopadhyay, S., Jullien, G.A., and Bayoumi, M., 1986. "Systolic arrays over finite rings with applications to digital signal processing." Systolic Array Workshop, Oxford, pp. 123-131
 14. Elleithy, K.M., Bayoumi, M., and Lee, K.P., 1989. "O(logN) Architectures for RNS Arithmetic Decoding." 9th IEEE Symposium on Computer Arithmetic, pp. 202-209
 15. Jullien, G.A., Miller, W.C., Grondin, R., Wang, Z., Zhang, D., Del Pup, L., and Bizzan, S., 1992. "WoodChuck: A Low-Level Synthesizer for Dynamic Pipelined DSP Arithmetic Logic Blocks." 1992 IEEE Int. Symp. on Circuits and Systems, San Diego, pp. 176-179
 16. Yuan, J. and Svensson, C., 1989. "High-Speed CMOS Circuit Technique." IEEE. J. Solid-State Circuits, **24**, pp. 62-70.
 17. Wigley, N.M., Jullien, G.A., Reaume, D., and Miller, W.C., 1991. "Small Moduli

- Replications in the MMRNS." 10th IEEE Symposium on Computer Arithmetic, Grenoble, pp. 92-99
18. Zhang, C.N., Shirazi, B., and Yun, D.Y.Y., 1987. "Parallel Designs for Chinese Remainder Conversion." International Conference on Parallel Processing, Chicago, pp. 557-559
 19. Griffin, M., Taylor, F., and Sousa, M., 1988. "New Scaling Algorithms for the Chinese Remainder Theorem." 22nd Asilomar Conference on Signals, Systems and Computers, pp. 375-378
 - 20*. S. Bandyopadhyay, G.A.Jullien and A. Sengupta, 1987 "A systolic architecture for implementing the Chinese Remainder Theorem", International Conference on Parallel Processing, Chicago, pp 924-931
 21. S. Bandyopadhyay, G. A. Jullien and A. Sengupta, 1988 "A systolic array for fault tolerant digital signal processing using a residue number system approach", International Conference on Systolic Arrays, pp 577-586

Acknowledgements:

This research has been supported, in part, by operating grants awarded to S. Bandyopadhyay and G. A. Jullien by Natural Sciences and Engineering Research Council of Canada.

Contact Person : S. Bandyopadhyay

Dr. S. Bandyopadhyay

School of Computer Science

University of Windsor

* This paper was inadvertently omitted from Conference Proceedings. Copies of the paper may be obtained from the first author.

Windsor, Ont, Canada, N9B 3P4

Tables and Figures

Stage i	Operation		
$i = 1, 2$	$(CP_i, R1_i, R2_i) = [(R1_{i-1}, R2_{i-1}) + Ai]$		
$i = 3$	$CP1$	$CP2$	Operation
	0	0	$CP3 = 0$ $R1_3 = R1_2$ $R2_3 = R2_2$
	0	1	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + (2^p - m)]$
	1	0	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + (2^p - m)]$
	1	1	$(CP3, R1_3, R2_3) = [(R1_2, R2_2) + 2.(2^p - m)]$
$i = 4, 5$	$CP(i-1)$		Operation
	0		$CP_i = 0$ $R1_i = R1_{i-1}$ $R2_i = R2_{i-1}$
	1		$(CP_i, R1_i, R2_i) = [(R1_{i-1}, R2_{i-1}) + (2^p - m)]$

Table 1 The 5-stage algorithm for Carry-Save RNS addition

Stage	Partial Sums and the carry
1	(000,110, 1)
2	(011,000, 1)
3	(101,100, 0)
4	(101,100, 0)
5	(101,100, 0)

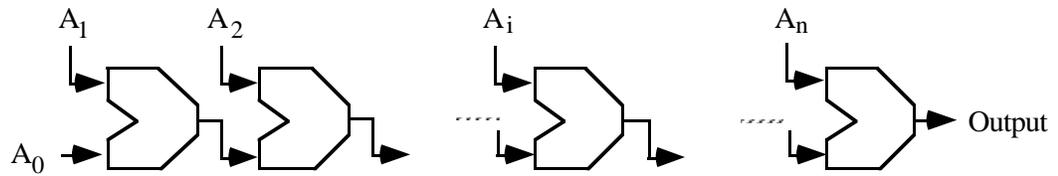
Table 2

W								W											W	W	
F					t		T	F	f							t		T		F	
				W	W	W								W	W	W	W				
F			t	T			F	T	t				f	F			t	T		T	
		W	W				W				W	W	W					W	W		
W		T		W		t	T	W		f	F			T		t		F		T	
	W	W				W	W			W		W	W			W	W		W		
T	T		F	t	T	T		F	T	f	F	T	F	t	T		F	f	T	F	T
			W	W				W	W	W		W									
W	F				T	W			W				W		W	t	T		F	F	T
W	W				W	F	f		T			f	F	t	T		W		W	W	W
W	W	W	W	W	W				W	W	W	W	W	W			W	W	W	W	
			F					f	F					F		f			F		

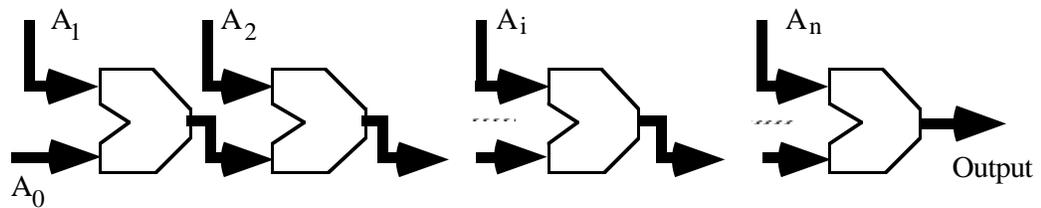
Table 3 Merged trees for the stage 3 switching function

Method	4 moduli, each 8 bits or less	6 moduli, each 6 bits or less	8 moduli, each 5 bits or less
Vu	54.71	57.45	55.66
Zhang	136.96	213.11	319.55
New method	19.96	39.92	59.88

Table 4 Silicon Area Comparison



a) Each adder block has 2, p-bit inputs and generates one p-bit output



b) Each adder block has 2 pairs of p-bit inputs and generates one pair of p-bit outputs

Fig. 1 Linear array of carry save adders

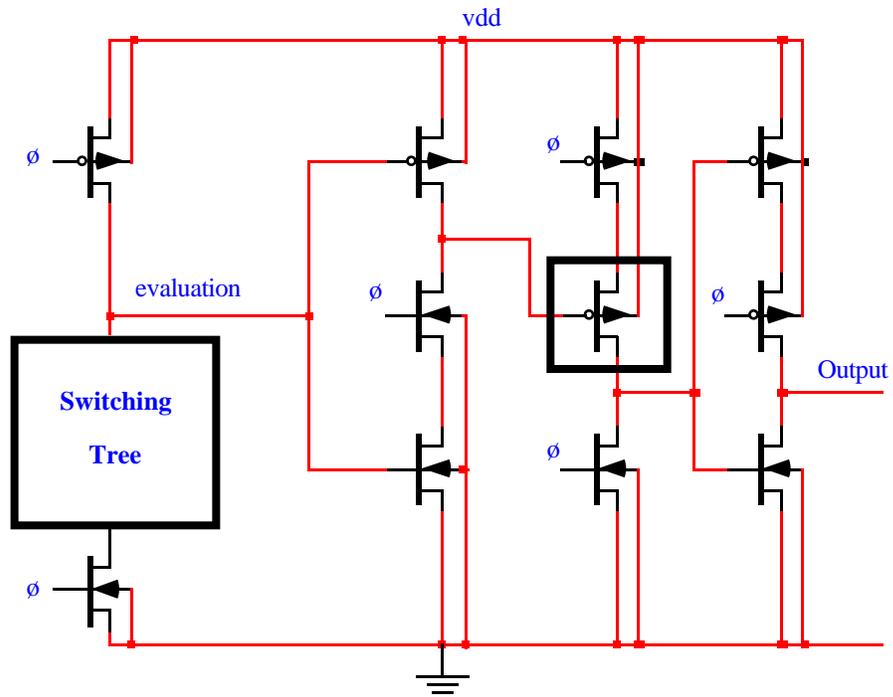


Fig. 2 Pipelining latch with single embedded tree

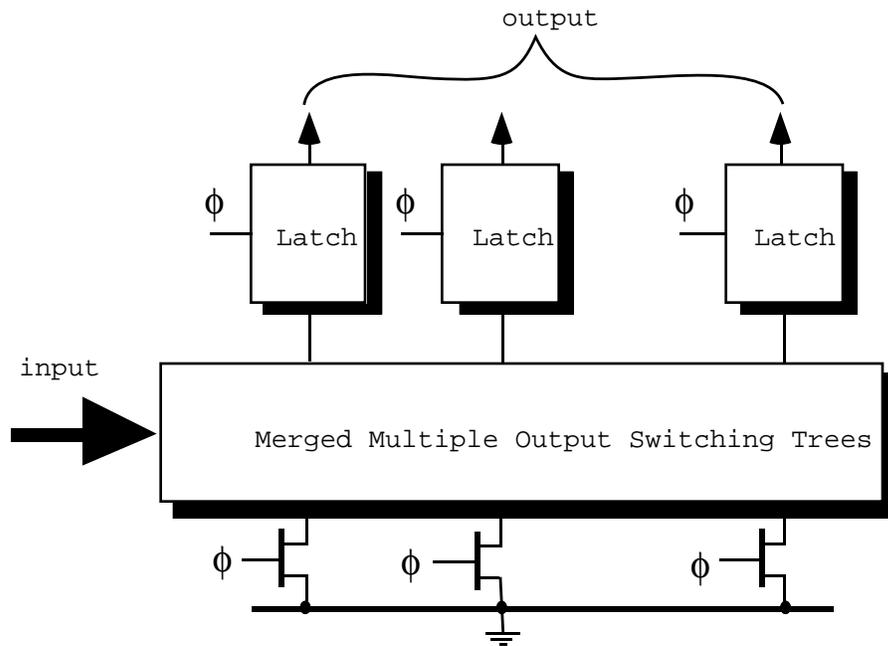


Fig. 3 A multiple output merged structure

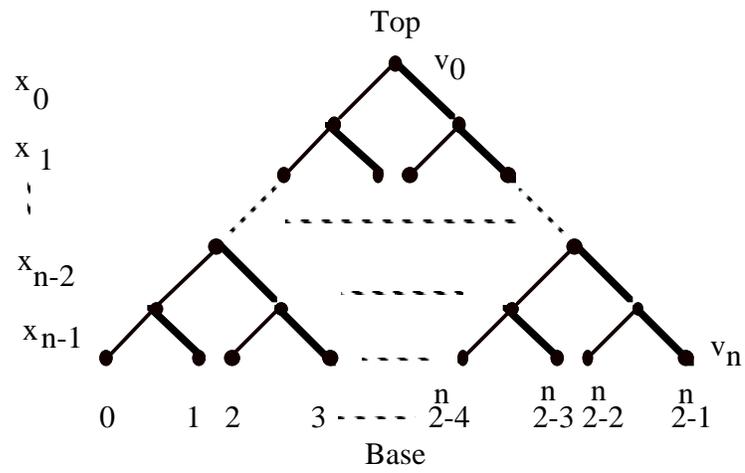


Fig 4 A Full Binary Tree

p-channel inverter strip	
Transistor Block	D-Latch
	D-Latch
	D-Latch
n-channel inverter strip	Buffer

Fig. 5 Block Image

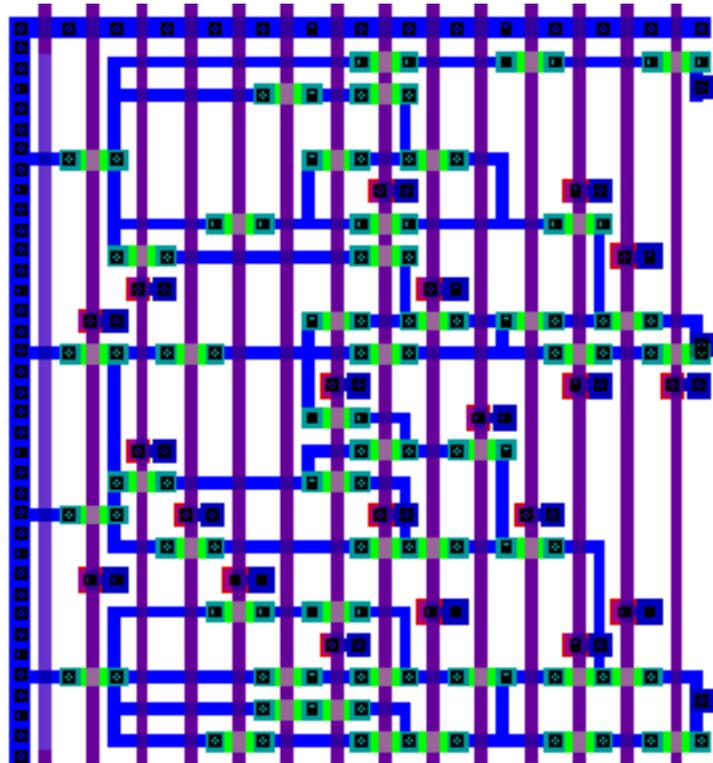


Fig. 6 Transistor block for Stage 3 merged trees

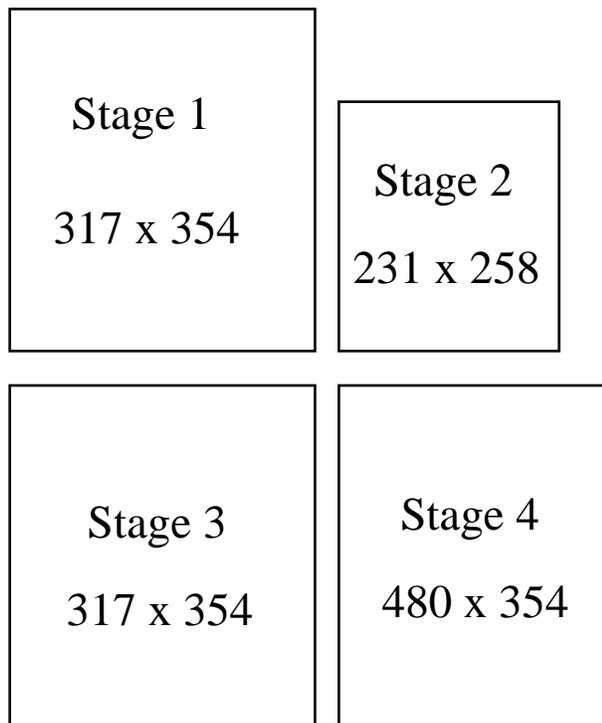


Fig. 7 Area comparison between the four switching functions

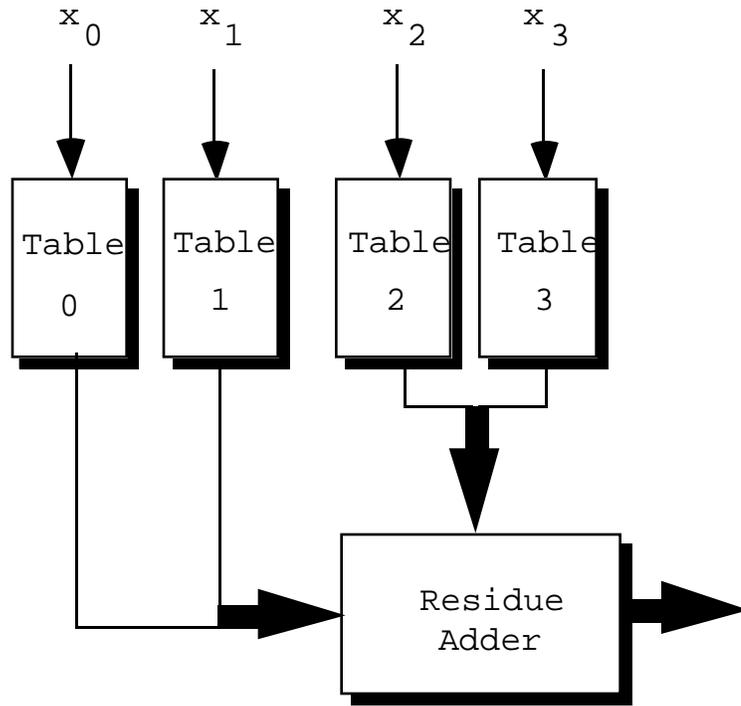


Fig. 8 Implementing the CRT