High Speed Parallel-Prefix Modulo 2^n+1 Adders for Diminished-One Operands

H. T. Vergos[1,2], C. Efstathiou[3] & D. Nikolos[1,2]
[1] *Computer Engineeing & Informatics Dept., University of Patras, 26500 Greece*
[2] *Computer Technology Institute, 3 Kolokotroni Str., 262 21 Patras, Greece*
[3] *Department of Informatics, TEI of Athens, Ag. Spyridonos St., 12210 Egaleo, Greece*
e-mail : vergos@cti.gr, cefsta@teiath.gr, nikolosd@cti.gr

Abstract

We present a new methodology for designing modulo 2^n+1 adders with operands in the diminished-one number system. The proposed methodology leads to parallel-prefix adder implementations. Both an analytical model and VLSI implementations in a standard-cell technology are utilized for comparing the adders designed following the proposed methodology against the existing solutions. Our results indicate that the proposed parallel-prefix adders are considerably faster than any other already known in the open literature and as fast as the corresponding modulo 2^n and modulo 2^n-1 adders.

1. Introduction

Modulo arithmetic has been used in digital computing systems for various purposes for many years. In particular modulo 2^n+1 arithmetic appears to play an important role in many algorithms.

High performance digital signal processing (DSP) systems often make use of the Residue Number System (RNS) [1-4]. In a RNS based application every number X is represented by a sequence of residues $(X_1, X_2, \ldots, X_M)$ where $X_i = X \bmod p_i$. The p_is, $1 \le i \le M$, comprise the base of the RNS and are pair-wise relative prime integers. A two operand RNS operation, suppose $\blacklozenge$, is defined as $(Z_1, Z_2, \ldots, Z_M) = (X_1, X_2, \ldots, X_M) \blacklozenge (Y_1, Y_2, \ldots, Y_M)$, where $Z_i = (X_i \blacklozenge Y_i) \bmod p_i$. For most RNS applications $\blacklozenge$ is either addition, subtraction or multiplication. Since the computation of Z_i only depends upon X_i, Y_i and p_i, each Z_i is computed in parallel in a separate arithmetic unit, often called *channel*. Moduli choices of the form $\{2^n-1, 2^n, 2^n+1\}$ have received significant attention because they offer very efficient circuits in the area x time^2 product sense [5]. Addition in such systems is performed using three channels, that in fact are a modulo 2^n-1 (equivalently one's complement), a modulo 2^n and a modulo 2^n+1 adder [1, 2]. The addition delay in an RNS application which uses the above moduli, is dictated by the modulo 2^n+1 channel. The latter means that if we can cut down the time required for modulo 2^n+1 addition we also cut down the addition time in a RNS application.

Modulo 2^n+1 adders are also utilized as the last stage adder of modulo 2^n+1 multipliers. Modulo 2^n+1 multipliers find applicability in :

- Pseudorandom number generation : special cases of the linear congruential sequence [6] use modulo 2^n+1 multiplication to obtain reasonably long sequence of pseudorandom numbers
- Cryptography : for attaining the desirable statistical independence between ciphertext and plaintext [7-9]
- In the Fermat number transform which is an effective way to compute convolutions because of its easy VLSI implementation and its lack of round off errors [10].

Leibowitz [11] proposed the *diminished-one* coding system, which is not only suitable for Fermat numbers, but also for general moduli of the form 2^n+1. Computations over modulo 2^n+1 rings, based on these numbers, is also adopted for many residue number system implementations [12, 13]. In the diminished-one system each number X is represented by $X^* = X - 1$. The representation of 0 is treated in a special way. Therefore, the adders that implement the diminished–one modulo 2^n+1 addition are combinational circuits accepting n bits wide operands.

Efficient VLSI implementations of modulo 2^n+1 adders for the diminished-one number system have recently been presented in [14, 15]. The adders presented in [14, 15] although fast are, according to the comparison presented in [14] still, slower than the fastest modulo 2^n adders or the fastest modulo 2^n-1 adders presented in [16]. Therefore their use in a RNS application would still limit the performance of the system.

In this paper we propose a new method for designing parallel-prefix modulo 2^n+1 adders. We also show using both an analytical model as well as VLSI implementations that the proposed parallel-prefix design methodology leads to considerably faster adder implementations than those presented in [14] and as fast as the modulo 2^n or the fastest

0-7695-1150-3/01 $10.00

modulo 2^n-1 architecture [16].

In the next Section we revisit the basics of speeding up the addition process. The derivation of the proposed here architecture is presented in Section III. Comparative results that show the time efficiency of the proposed architecture against the existing solutions are presented in Section IV. Some conclusions are given in the last section.

2. Preliminaries

In order to speed up the addition operation, the carry computation time should be minimized. One solution is to use carry look-ahead (CLA) adders [2, 17].

However, when the operand length is large the number of inputs to the high order gates of the carry computation unit also becomes quite large. In current VLSI technology gates with a large number of inputs are either not available or too slow. Therefore such circuits are designed as a tree of gates with less inputs, leading to a circuit with more logic levels and therefore of increased delay. Consequently, in the case of wide operands it is profitable to design the carry computation unit with more than one levels of CLA [17]. Under this approach at each level of the carry computation unit the inputs are divided into groups, a smaller CLA unit is employed for each group and collective CLA units are introduced for between groups carry computations. In the general case of a multi-level CLA adder both the number of levels and the number of groups in each level needs to be investigated for each implementation technology for reaching the design that best balances performance and implementation area.

A special form of CLA adders, well known as parallel–prefix form, can be derived if carry computation in binary addition is treated as a prefix problem [18]. In a prefix problem n inputs (suppose $x_{n-1}, x_{n-2}, \ldots, x_0$) and an arbitrary associative operator o are used for computing n outputs, suppose $y_{n-1}, y_{n-2}, \ldots, y_0$, according to the relation $y_i = x_i \, o \, x_{i-1} \, o \, \ldots \, o \, x_0$, for i = 0, …, n-1. Let $A = a_{n-1}a_{n-2} \ldots a_1a_0$ and $B = b_{n-1}b_{n-2} \ldots b_1b_0$ be two n-bit numbers and $S = s_{n-1}s_{n-2} \ldots s_1s_0$ their sum. Carry computation is transformed into a prefix computation if the associative operator o is defined according to [19] as :

$$(g_m, p_m) \, o \, (g_k, p_k) = (g_m + p_m \cdot g_k, p_m \cdot p_k),$$

where :

- $g_i = a_i \cdot b_i$, is the carry generate and
- $p_i = a_i + b_i$. is the carry propagate term. (Note that p_i may also be defined as $p_i = a_i \oplus b_i$, where $\oplus$ denotes the exclusive-OR operation, but since this leads to somewhat slower implementations, we will not adopt it in this work)

Then the carries are computed as $c_i = G_i$, where G_i is the first member of the group relation (assuming that carry input $c_{in} = 0$):

$$(G_i, P_i) = \begin{cases} (g_0, p_0), & \text{if } i = 0 \\ (g_i, p_i) \, o \, (G_{i-1}, P_{i-1}), & \text{if } 1 \le i \le n-1 \end{cases} \quad (1)$$

After the computation of the carries the sum bits are given by $s_i = h_i \oplus c_{i-1}$, where $h_i = a_i \oplus b_i$ is the half-sum.

Usually the o operation on a pair of (g_x, p_x) terms is represented as a node (see Fig. 1) and a whole carry computation unit is represented as a tree structured interconnection of such nodes. Several tree structures have been proposed in the past [19-21]. Figs. 2 and 3 present for n = 8 the tree structures proposed by Sklansky [20] and Kogge–Stone [21] respectively, that are the fastest among the already proposed. The gate level implementation of the nodes is given in Fig. 1. The insertion of the buffering nodes, ○, is not mandatory. The adders that result following a proposed tree structure feature layout regularity, but each structure has distinct implementation area, speed and fan-out characteristics. For example, adders with a Sklansky prefix structure require less implementation area but have increased fan-out compared to adders with a Kogge-Stone prefix structure. A full comparison among the already known parallel-prefix adders can be found in [15].

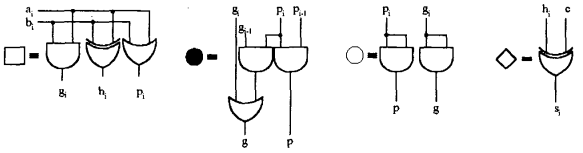

Fig. 1. Logic operators and their implementations

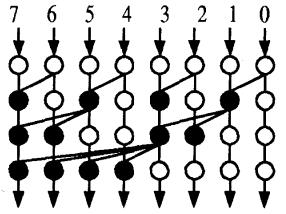

Fig. 2. Sklansky Structure

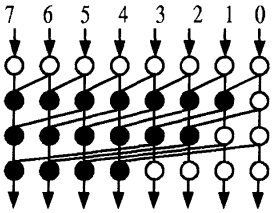

Fig. 3. Kogge–Stone Structure.

Adders with a carry input signal $c_{in} = c_{-1}$ can also be designed with a simple extension of the parallel-prefix architecture. Let $\hat{G}_i$, $\hat{P}_i$ denote the new generate and propagate functions respectively, assuming a carry input $c_{in} = c_{-1} \in \{0, 1\}$, where

$$(\hat{G}_i, \hat{P}_i) = \begin{cases} (g_0 + p_0 \cdot c_{-1}, p_0), & \text{if } i = 0 \\ (g_i, p_i)\ o\ (\hat{G}_{i-1}, \hat{P}_{i-1}), & \text{if } 1 \le i \le n-1. \end{cases} \quad (2)$$

and $\hat{c}_i = \hat{G}_i$ for $0 \le i \le n-1$. Then, as it has been proven in [16] the generate and propagate signals for an adder with a carry input can be derived by those of an adder without a carry input by the relation :

$$(\hat{G}_i, \hat{P}_i) = (G_i + P_i \cdot c_{-1}, P_i) \quad (3)$$

Relation (3) implies that by adding a single row of logic blocks to the output of the carry computation unit (irrespectively of the architecture used for the implementation of the latter) we can construct an adder with a carry input. Such a modification is shown in Fig. 4.

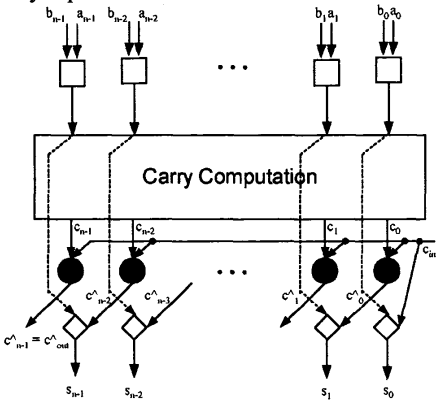

Fig. 4. Parallel-prefix adder structure with c_{in}

For modulo 2^n+1 operations the diminished-one number system is often used. In this system the value 0 is not used or is treated separately (for example using an additional zero-indication bit). We will hereafter denote with X* the representation of X in the diminished-one number system, that is, X* = X – 1. If S is the sum of A and B then (also stated in [14, 15]) :

$$A + B = S \Leftrightarrow (A^* + 1) + (B^* + 1) = S^* + 1 \Leftrightarrow$$

$$\Leftrightarrow A^* + B^* + 1 = S^* \Rightarrow$$

$$(A^*+B^*+1) \bmod (2^n+1) = \begin{cases} A^* + B^* + 1 - (2^n+1), \\ \text{if } A^* + B^* + 1 \ge 2^n+1 \\ A^* + B^* + 1 \text{ otherwise} \end{cases} \Leftrightarrow$$

$$(A^*+B^*+1) \bmod (2^n+1) = \begin{cases} A^* + B^* - 2^n, & \text{if } A^* + B^* \ge 2^n \\ A^* + B^* + 1 \text{ otherwise} \end{cases} \Leftrightarrow$$

$$(A^*+B^*+1) \bmod (2^n+1) = \begin{cases} (A^*+B^*) \bmod 2^n, & \text{if } A^*+B^* \ge 2^n \\ A^* + B^* + 1 \text{ otherwise} \end{cases}$$

The last relation reveals that a diminished-one modulo 2^n+1 adder can be implemented by incrementing the sum when the carry output $c_{out} = 0$, or equivalently if we connect the carry output of the carry computation unit in Fig. 4 to the carry input via an inverter. Such a design has been reported in [14, 15] and is presented in Fig. 5. When parallel-prefix structures are used for the carry computation unit, the resulting adders [14, 15] are the fastest diminished-one modulo 2^n+1 adders known in the literature.

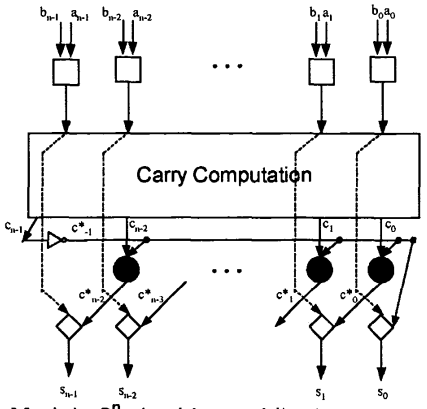

Fig. 5. Modulo 2^n+1 adder architecture proposed in [14, 15].

It is obvious that following the above design method the resulting diminished-one modulo 2^n+1 adders will always be slower than the corresponding parallel-prefix modulo 2^n adders and the fastest parallel-prefix modulo 2^n-1 adders [16] because :

- of the extra stage of ● operators that is required and
- the re-entering carry has a large fan-out.

To this end, in the next section we present new and faster parallel-prefix implementations derived by recirculating the carries in each level of the carry computation unit instead of having a final stage of re-entering carry.

We should note that all diminished-one adders suffer from the double zero output meaning problem, that is, a zero output may either represent an actual zero output (that is, an addition with a result of one in a normal representation) or an erroneous representation of zero. However, erroneous zero output results only when the two inputs are complementary. Therefore, this situation can be easily detected using a simple combinational circuit.

3. Novel modulo 2^n+1 adder design

Let X' denote the complement of X. The last stage of ● operators in Fig. 5, accepts the re-entrant carry $c^*_{-1} = G'_{n-1}$ and produce the modulo 2^n+1 carries $c^*_i = \hat{G}_i = G_i + P_i \cdot G'_{n-1}$, for $0 \le i \le n-2$. As we have mentioned earlier, this design apart from adding an extra logic operator stage also has the disadvantage that the re-entering carry has a fan-out of n. Therefore in the sequel we utilize the idea of carry recirculation in each prefix level, that was introduced in [16], for transforming the computation of the carries c^*_i of the modulo 2^n+1 adder, for $-1 \le i \le n-2$, in a parallel-prefix computation problem.

We define the complement of (G, P) denoted by (G, P)' to be equal to (G', P) and the group generate and group propagate functions $G_{a,b}$ and $P_{a,b}$ for the group of bits a, a-1, ..., b, with a > b as $(G_{a,b}, P_{a,b}) = (g_a, p_a)\ o\ (g_{a-1}, p_{a-1})\ o \ldots o\ (g_b, p_b)$.

The novel parallel - prefix modulo 2^n+1 adder design method, that we propose in this paper, is based on the following three theorems :

Theorem 1.

Let c^*_i, with $-1 \le i \le n-2$, be the carries of the modulo 2^n+1 addition and

$$(G^*_i, P^*_i) = \begin{cases} (G_{n-1}, P_{n-1})', & \text{if } i = -1 \\ (g_i, p_i)\ o\ (G^*_{i-1}, P^*_{i-1}), & \text{if } 0 \le i \le n-2, \end{cases}$$

where G_{n-1}, P_{n-1} are computed according to relation (1). Then, $c^*_i = G^*_i$.

Proof.

We will use induction for the proof of the Theorem.

- For i = -1 we have that $(G^*_{-1}, P^*_{-1}) = (G_{n-1}, P_{n-1})'$ or $(G^*_{-1}, P^*_{-1}) = (G'_{n-1}, P_{n-1})$. Therefore $c^*_{-1} = G'_{n-1} = G^*_{-1}$.
- Assume that the relation holds for i = k, which means that $c^*_k = G^*_k$.
- Then for i = k+1, we have that :

$$\begin{aligned}(G^*_{k+1}, P^*_{k+1}) &= (g_{k+1}, p_{k+1})\ o\ (G^*_k, P^*_k) \\ &= (g_{k+1}, p_{k+1})\ o\ (c^*_k, P^*_k) \\ &= (g_{k+1} + p_{k+1}c^*_k, p_{k+1}P^*_k).\end{aligned}$$

The above relation leads to $G^*_{k+1} = g_{k+1} + p_{k+1}c^*_k$. Since $c^*_{k+1} = g_{k+1} + p_{k+1}c^*_k$, we get $c^*_{k+1} = G^*_{k+1}$. ■

Theorem 2.

$(G^*_i, P^*_i) = (G_i, P_i)\ o\ (G_{n-1,i+1}, P_{n-1,i+1})'$

Proof.

According to theorem 1 we have

$$\begin{aligned}(G^*_i, P^*_i) &= (G_i, P_i)\ o\ (G_{n-1}, P_{n-1})' = (G_i, P_i)\ o \\ &\quad ((G_{n-1,i+1}, P_{n-1,i+1})\ o\ (G_i, P_i))' = \\ &= (G_i, P_i)\ o\ (G_{n-1,i+1} + P_{n-1,i+1}G_i, P_{n-1,i+1}P_i)' = \\ &= (G_i, P_i)\ o\ ((G_{n-1,i+1} + P_{n-1,i+1}G_i)', P_{n-1,i+1}P_i) = \\ &= (G_i + P_i(G_{n-1,i+1} + P_{n-1,i+1}G_i)', P_iP_{n-1,i+1}P_i) = \\ &= (G_i + P_i\,G'_{n-1,i+1}(P'_{n-1,i+1} + G'_i),\ P_iP_{n-1,i+1}) = \\ &= ((G_i + P_i\,G'_{n-1,i+1})\,(G_i + P'_{n-1,i+1} + G'_i),\ P_iP_{n-1,i+1}) = \\ &= (G_i + P_i\,G'_{n-1,i+1},\ P_iP_{n-1,i+1}) = \\ &= (G_i, P_i)\ o\ (G'_{n-1,i+1},\ P_{n-1,i+1}) = \\ &= (G_i, P_i)\ o\ (G_{n-1,i+1},\ P_{n-1,i+1})'\end{aligned}$$ ■

Theorem 3.

Suppose that $(G_x, P_x) = (G, P)\ o\ (G_{i,m}\ P_{i,m})'$ and that $(G_y, P_y) = (G, P)\ o\ (p'_i, g'_i)\ o\ (G_{i-1,m}, P_{i-1,m})'$, with i > m. Then $G_x = G_y$.

Proof.

Since $(G_x, P_x) = (G, P)\ o\ (G_{i,m}\ P_{i,m})' = (G, P)\ o\ (G'_{i,m}\ P_{i,m})$ we get that :

$$G_x = G + PG'_{i,m} \qquad (4)$$

Since $(G_y, P_y)=(G, P)\ o\ (p'_i, g'_i)\ o\ (G_{i-1,m}, P_{i-1,m})'=(G, P)\ o\ (p'_i, g'_i)\ o\ (G'_{i-1,m}, P_{i-1,m})$, we get that :

$$\begin{aligned}G_y &= G + P(p'_i + g'_iG'_{i-1,m}) = G + P(g'_i\,p'_i + g'_iG'_{i-1,m}) = G + \\ &P(g'_i(p'_i + G'_{i-1,m})) = \\ &G + P(g_i + p_i\,G_{i-1,m})' = G + PG'_{i,m}. \qquad (5)\end{aligned}$$

From (4), (5) we get the required $G_x = G_y$. ■

Theorem 2 equivalently implies that the relation $(G^*_i, P^*_i) = (g_i, p_i)\ o \ldots o\ (g_0, p_0)\ o\ ((g_{n-1}, p_{n-1})\ o \ldots o\ (g_{i+1}, p_{i+1}))'$ holds for $0 \le i \le n-2$, that is, the carries in a modulo 2^n+1 adder can be computed using a prefix structure in which the carry at each position i does not only depend on bits i to 0 but also on the bits n-1 through i+1. This can be achieved by re-circulating the carries at each prefix level instead of having a single final stage for the re-entrant carry, as proposed in [16]. For achieving carry re-circulation at each prefix level more logic operators must be added to a Kogge-Stone prefix structure and the outputs of the highest order 2^{m-1} operators of stage m-1 need to be driven to the lowest order 2^{m-1} operators of stage m.

The fastest parallel-prefix modulo 2^n and modulo 2^n-1 adders are capable of computing the carries within $m=\log_2 n$ prefix levels. For not delaying the addition operation in a RNS environment, the proposed adders should also be able to compute the carries within $\log_2 n$ prefix levels. However, the equations in the form produced by Theorem 2 can not always be implemented in $\log_2 n$ prefix levels. To overcome this, one needs to apply Theorem 3 to the terms $(G_{n-1,i+1},\ P_{n-1,i+1})'$ j times recursively, until n-1-j-i equals to a power of 2. That is although Theorem 2 defines the carry at each bit position i $(-1 \le i \le n-2)$ as :

$c^*_i = G^*_i$, where $(G^*_i, P^*_i) = (G_i, P_i)\ o\ (G_{n-1,i+1}, P_{n-1,i+1})'$, efficient parallel-prefix implementations are derived by transforming the above computation in :

$$(G^*_i, P^*_i) = (G_i, P_i)\ o\ (p'_{n-1}, g'_{n-1})\ o \ldots o\ (p'_{n-j+1}, g'_{n-j+1})\ o\ (G_{n-1-j,i+1}, P_{n-1-j,i+1})'$$

The latter relations can be implemented in $\log_2 n$ levels using a prefix structure which needs more prefix logic operators as well as modifications of some of the existing in order to produce except the normal terms their complements.

In the general case of n bits wide operands, the carries for a diminished-one modulo 2^n+1 adder can be computed in $\log_2 n$ stages, where the first $(\log_2 n - 1)$ stages are comprised by $3\frac{n}{2} - 1$ operators and the last by n operators.

Although the hardware overhead that the above described modifications impose may at first seem large, many of the extra logic operators are shared among the computation of several carries and the hardware overhead of the modification for the complement generation is too small.

Example.

Consider two 8-bit operands $A = a_7a_6a_5a_4a_3a_2a_1a_0$ and B =

$b_7b_6b_5b_4b_3b_2b_1b_0$ in the diminished-one number representation. For constructing a parallel-prefix modulo 257 adder for them, from Theorem 2 we get the following equations :

$c_{-1} = ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0))'$

$c_0 = (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1))'$

$c_1 = (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2))'$

$c_2 = (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3))'$

$c_3 = (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4))'$

$c_4 = (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5))'$

$c_5 = (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6))'$

$c_6 = (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7))'$

The carries of the fastest modulo 256 parallel–prefix adders can be computed in 3 prefix levels. Since we want our adders to be as fast as the modulo 256 ones, we should be able to compute c_{-1} up to c_6 within 3 parallel-prefix levels. Consider however the equation for c_0. Since this equation has 8 terms that need to be associated by the use of operator o which treats its left and right operands distinctly, it is clear that the computation of c_0 as indicated by the above equation can not be achieved within 3 parallel-prefix levels. This is obvious since for computing $(G_{7,1}, P_{7,1})'$ the 3 prefix levels are exhausted and a 4th level is required. In such cases Theorem 3 should be used recursively. Applying Theorem 3 in the c_0 computation case, we get :

$$c_0 = (g_0,p_0)\ o\ (p'_7,g'_7)\ o\ (p'_6,g'_6)\ o\ (p'_5,g'_5)\ o\ ((g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1))'$$

The latter expression of c_0 can be implemented within 3 parallel-prefix levels assuming that more operators as well as small modifications of those existing in a Kogge-Stone structure are introduced as follows :

- The complements of g_7, p_7, g_6, p_6, g_5 and p_5 must be formed.
- At the first prefix level more operators are introduced as follows :
  - ❖ An operator for computing $(g_0,p_0)\ o\ (p'_7,g'_7)$.
  - ❖ An operator for computing $(p'_6,g'_6)\ o\ (p'_5,g'_5)$.
- At the second prefix level one more operator that computes the term $(g_0,p_0)\ o\ (p'_7,g'_7)\ o\ (p'_6,g'_6)\ o\ (p'_5,g'_5)$ is required along with a slight modification of the operator that produces $(G_{4,1}, P_{4,1})$ in order to also produce $(G_{4,1}, P_{4,1})'$.
- At the last prefix level one more operator that produces c_0 is added.

Applying Theorem 3 where necessary in the above carry equations we can get the following set of equations that can all be implemented within 3 parallel prefix levels :

$c_{-1} = ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0))'$

$c_0 = (g_0,p_0)\ o\ (p'_7,g'_7)\ o\ (p'_6,g'_6)\ o\ (p'_5,g'_5)\ o\ ((g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1))'$

$c_1 = (g_1,p_1)\ o\ (g_0,p_0)\ o\ (p'_7,g'_7)\ o\ (p'_6,g'_6)\ o\ ((g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2))'$

$c_2 = (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ (p'_7,g'_7)\ o\ ((g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3))'$

$c_3 = (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4))'$

$c_4 = (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ (p'_7,g'_7)\ o\ ((g_6,p_6)\ o\ (g_5,p_5))'$

$c_5 = (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ ((g_7,p_7)\ o\ (g_6,p_6))'$

$c_6 = (g_6,p_6)\ o\ (g_5,p_5)\ o\ (g_4,p_4)\ o\ (g_3,p_3)\ o\ (g_2,p_2)\ o\ (g_1,p_1)\ o\ (g_0,p_0)\ o\ (g_7,p_7)'$

In the above equations it is easy to see that the extra operator introduced for computing $(g_0,p_0)\ o\ (p'_7,g'_7)$ is shared among the computation of c_0, c_2 and c_4. The modulo 257 adder that can be devised by the above equations is presented in Fig. 6. Modified operators are indicated by gray color.

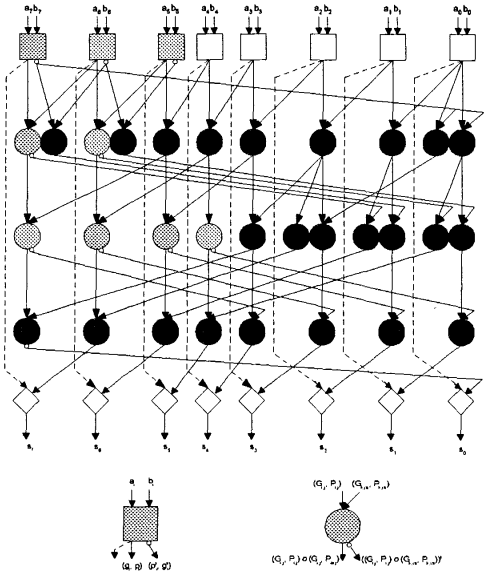

Fig. 6. Proposed diminished-one modulo 257 adder.

4. Comparisons

In this section, at first, we compare the proposed adders against those proposed in [14, 15] using both an analytical model as well as VLSI implementations. We use the notation SKL and KST for diminished-one modulo 2^n+1 adders with the architecture proposed in [14, 15] and presented in Fig. 5 and a Sklansky or a Kogge-Stone prefix

carry computation structure respectively.

We will at first make use of the analytical model originally presented in [22], that was also used in [14 - 16], under the notation "unit-gate model", for comparing the proposed adders against SKL and KST adders. This model assumes that each gate excluding exclusive-OR, counts as one elementary gate for both area and delay. An exclusive-OR gate counts for two elementary gates for both area and delay. The model ignores fan-in and fan-out, therefore the validation of the estimates that it produces will be later carried out by CMOS static implementations.

Table I. Adder Area and Delay Model Estimations

Adder	Area	Delay
Modulo 2^n	$\frac{3}{2}n\log n + 5n$	$2\log n + 3$
Fastest Modulo 2^n-1 [16]	$3n\log n + 5n$	$2\log n + 3$
SKL	$\frac{3}{2}n\log n + 8n - 1$	$2\log n + 5$
KST	$3n\log n + 7n$	$2\log n + 5$
Proposed	$\left(\frac{9n}{2} - 3\right)\log n + 3n - 2$	$2\log n + 3$

In Table I we present the area and delay estimates using this model as a function of the word length n. We have also included results for the fastest modulo 2^n and modulo 2^n-1 [16] adder. For the derivation of the area estimates the ○ buffering nodes in the carry computation prefix structure of Figs 2 and 3, have not been taken into account.

Considering the delay, Table I does not only reveal that the proposed adders are faster than both SKL and KST adders, but that they can operate as fast as the fastest known modulo 2^n and modulo 2^n-1 adders, which makes them ideal for use in an RNS application. From Table I we can see that among the modulo 2^n+1 adders, SKL adders require the less area for their implementation. It should be noted however that the estimations produced by the adopted model do not take into account the area that may be required for buffer insertion needed to alleviate the unlimited fan-out problem of Sklansky prefix structures.

For more realistic evaluation, the proposed as well as SKL and KST adders were described in HDL for n=4, 8, 16 and 32. For direct comparisons with the results presented in [16], we mapped our designs to the AMS CUB implementation technology (0.6 μm, 2-metal layer, 5.0 V) using the Design Analyzer® tool of Synopsys Inc. Each design was then recursively optimized for speed until the tool's algorithm was unable to provide a faster design. As a last stage of each recursive run the tool was instructed to recover as much area as possible. Table II lists the obtained results. The delay results are based on the assumption of worst case process parameters and are expressed in ns, whereas area results are expressed in mils^2.

Table II. Area and delay results of static CMOS implementations

n	Architecture	Area	Delay
4	SKL	66.7	2.70
	KST	87.2	2.69
	Proposed	84.0	2.02
8	SKL	183.2	3.87
	KST	189.9	3.82
	Proposed	199.1	3.20
16	SKL	349.6	4.64
	KST	426.3	4.58
	Proposed	694.5	3.85
32	SKL	706.8	5.80
	KST	955.6	5.79
	Proposed	1356.4	4.79

Table II indicates that the proposed are the fastest diminished-one modulo 2^n+1 adders. Moreover, the performance difference becomes larger in favor of the proposed adders as n becomes larger. On the average of the examined cases the proposed are approximately faster by 19% than SKL or KST adders. For reaching the fastest implementations our proposed design methodology requires on the average 31% and 17% more implementation area over SKL and KST adders.

Table III. Area-Time Constraint Driven Optimization Results

n	Area	Delay
4	58.9	2.57
8	155.8	3.78
16	425.2	4.58
32	1005.0	5.78

As one can observe in Table II, in all examined cases the proposed methodology leads to faster than SKL and KST adders. However, these fastest implementations require more area than the fastest implementations of SKL and KST adders. Therefore, it is interesting to examine whether when we restrict the area of the proposed adders to that of the fastest among SKL and KST adders, the proposed adders would still offer better performance. In Table III we present area and delay results obtained by instructing the synthesis tool to find the smaller implementation of the proposed adders that offers at least the performance of the fastest SKL or KST adders. Comparing the results of Tables II and III, we can see that in all but one case (in the n=32 case the tool gave a slightly faster implementation with a slightly larger area) the

proposed adders can lead to at least the same performance as the fastest among SKL and KST adders with smaller implementation area.

We have also described in HDL modulo 2^n adders with both a Sklansky and a Kogge – Stone adder, for n=8, 16 and 32 and mapped them to the above technology targeting the minimal possible delay. Table IV lists the results obtained for the fastest among them as well as results from Table II above and from Table III of [16]. These results indicate that the proposed adders apart from being the fastest adders for diminished-one modulo 2^n+1 addition, offer a delay close to that of the fastest modulo 2^n or modulo 2^n-1 adders. Therefore, the proposed adders are highly suitable for RNS applications.

Table IV. The proposed adders as part of an RNS

	Fastest Modulo 2^n		Modulo 2^n-1 [16]		Proposed Modulo 2^n+1	
n	**Area**	**Delay**	**Area**	**Delay**	**Area**	**Delay**
8	138.6	2.85	196.6	3.30	199.1	3.20
16	451.6	3.78	588.7	4.05	694.5	3.85
32	1150.1	4.55	1341.2	4.97	1356.4	4.79

5. Conclusions

In this paper we have presented a novel architecture for designing diminished-one parallel-prefix modulo 2^n+1 adders. Our architecture was derived by recirculating the carries in each level of the prefix structure, instead of the earlier proposed method of re-entering the final carry at an additional stage. Static CMOS implementations have shown that the proposed modulo 2^n+1 adders compare favorably with the other already known adder architectures. Moreover, the parallel-prefix modulo 2^n+1 adders proposed are as fast as the fastest modulo 2^n and modulo 2^n-1 adders, that is, highly applicable in RNS applications.

6. References

[1] M. A. Sonderstrand et. al., Residue Number System Arithmetic : Modern Applications in Digital Signal Processing, IEEE Press, New York, 1986.

[2] I. Koren, Computer Arithmetic Algorithms, Prentice–Hall, 1993.

[3] K. M. Elleithy & M. A. Bayoumi, "Fast and Flexible Architectures for RNS arithmetic decoding", IEEE Transactions on Circuits and Systems–II : Analog and Digital Signal Processing, vol. CAS-39, pp. 226 – 235, April 1992.

[4] M. A. Bayoumi et. al., "A Look-Up Table VLSI Design Methodology for RNS Structures used in DSP Applications", IEEE Transactions on Circuits and Systems, vol. CAS-34, pp. 604-616, June 1987.

[5] V. Paliouras and T. Stouraitis, "Novel High – Radix Residue Number System Multipliers and Adders", in Proc. of the 1999 IEEE International Symposium on Circuits and Systems VLSI (ISCAS '99), Orlando, FL, USA, pp. 451 – 454, 1999.

[6] D. H. Lehmer, in Proc. of the 2nd Symposium on Large – Scale Digital Calculating Machinery, Cambridge, MA : Harvard University Press, 1951, pp. 141 – 146.

[7] R. Zimmermann et. al., "A 177 Mb/s VLSI Implementation of the International Data Encryption Algorithm", IEEE Journal of Solid - State Circuits, vol. 29 (3), pp. 303 – 307, March 1994.

[8] A. Curiger, *VLSI Architectures for Computations in Finite Rings and Fields*, Ph.D. Thesis, Swiss Federal Institute of Technology (ETH), Zurich 1993.

[9] X. Lai and J. L. Massey, "A proposal for a new block encryption standard", presented at EUROCRYPT '90, Aarhus, Denmark, May 1990.

[10] Y. Ma, "A Simplified Architecture for Modulo (2^n+1) Multiplication", IEEE Transactions on Computers, Vol. 47, No. 3, March 1998.

[11] L. M. Leibowitz, "A simplified binary arithmetic for the Fermat number transform", IEEE Trans. Acoust. Speech, Signal Processing, Vol. ASSP-24, pp. 356-359, 1976.

[12] W. K. Jenkins, "The design of specialized residue classes for efficient recursive digital filter realization", IEEE Trans. Acoust. Speech and Signal Processing, Vol. ASSP-30, pp. 370-380, 1982.

[13] W. K. Jenkins, "Recent Advance in residue number techniques for recursive digital filtering", IEEE Trans. Acoust. Speech and Signal Processing, Vol ASSP-27, pp. 19 – 30, 1979.

[14] R. Zimmermann, "Efficient VLSI Implementation of Modulo $(2^n\pm1)$ Addition and Multiplication", in Proc. of 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, pp. 158–167, April 1999.

[15] R. Zimmermann, Binary Adder Architectures for Cell-Based VLSI and their Synthesis, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich 1997.

[16] L. Kalamboukas, et. al, "High-Speed Parallel-Prefix Modulo 2^n-1 Adders", IEEE Transactions on Computers, Special Issue on Computer Arithmetic, Vol. 49, No.7, pp. 673 - 680, July 2000.

[17] K. Hwang, Computer Arithmetic : Principles, Architecture and Design, John Wiley and Sons, 1979.

[18] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", Journal of the ACM, Vol. 27 (4), pp. 831–838, October 1980.

[19] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders", IEEE Trans. on Computers, Vol. C-31 (3), pp. 260–264, March 1982.

[20] J. Sklansky, "Conditional Sum Addition Logic", IRE Trans. Electron. Comput., EC-9 (6), pp. 226–231, June 1960.

[21] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Transactions on Computers, Vol. 22 (8), pp. 783–791, August 1973.

[22] A. Tyagi, "A Reduced–Area Scheme for Carry–Select Adders", IEEE Trans. on Computers, Vol. 42 (10), pp. 1163–1170, October 1993.