A O(1) Algorithm for Modulo Addition
KHALED M. ELLEITHY and MAGDY A. BAYOUMI

Abstract—A O(1) algorithm for large modulo addition for residue number system (RNS) based architectures is proposed. The addition is done in a fixed number of stages which does not depend on the size of the modulus. The proposed modulo adder is much faster than the previous adders and more area efficient. The implementation of the adder is modular and is based on simple cells which leads to efficient VLSI realization.

I. INTRODUCTION

Recently, the residue number system (RNS) is receiving increased attention due to its ability to support high-speed concurrent arithmetic [1]. Applications such as fast Fourier transform, digital filtering, and image processing utilize the high-speed RNS arithmetic operations; addition and multiplication, do not require the difficult RNS operations such as division and magnitude comparison. The technological advantages offered by VLSI have added a new dimension in the implementation of RNS-based architectures [2]. Several high-speed VLSI special purpose digital signal processors have been successfully implemented [3]-[5].

Modulo addition represents the computational kernel for RNS-based architectures. Subtraction is performed by adders using the additive inverse property [6]. Multiplication can be transformed into addition by several techniques [7]. Also, modulo addition is the basic element in the conversion from RNS to binary using the Chinese remainder theorem (CRT) [6]. Banerji [8] analyzed modulo addition in MSI technology. A VLSI analysis of modulo addition has been reported in [9]-[11]. In general, lookup tables and PLAs have been the main logical modules used when the data granularity is the word. It has been found that such structure is only efficient for small size moduli. For medium size and large moduli, bit-level structures are more efficient, where the data granularity is the bit [12].

In this paper, we present a modulo adder for medium size and large moduli. It is based on using a two-dimensional array of very simple cells (full adders). The modulo addition is performed in a fixed time delay independent of the size of the moduli.

II. RESIDUE NUMBER SYSTEM (RNS)

In RNS, an integer $X$ can be represented by $N$-tuple of residue digits

$X = (r_1, r_2, \ldots, r_N)$

where $r_i = |X|_m_i$, with respect to a set of $N$ moduli $\{m_1, m_2, \ldots, m_N\}$. In order to have a unique residue representation, the moduli must be pairwise relatively prime, that is:

$\text{GCD}(m_i, m_j) = 1,$ for $i \neq j$.

Then it is shown that there is a unique representation for each number in the range of $0 < X < \prod_{i=1}^{N} m_i = M$, where $N$ is the number of moduli.

The arithmetic operation on two integers $A$ and $B$ is equivalent to the arithmetic operation on its residue representation, that is:

$[A \cdot B]_m = (|A|/m_1 |B|/m_1 |A|/m_2 |B|/m_2 |A|/m_3 |B|/m_3 |\cdots)$

$|A|/m_N |B|/m_N$,

where "\cdots" can be addition, subtraction, or multiplication. It is desirable to convert binary arithmetic on large integers to residue arithmetic on smaller residue digits in which the operations can be parallelly executed, and there is no carry chain between residue digits.

2.1. The Modulo Addition

Generally, addition modulo $m$ has $2^n - m$ $(n = \lfloor \log m \rfloor)$ incorrect residue states. These states are in the range $\{m, 2^n - 1\}$ which may be called overflow states. The corrected residue numbers can be obtained by two methods; employing a binary adder or a correction table. In the first method, a constant $(2^n - m)$ is added to correct the overflow residue states (generalized end-round carry) as shown in Fig. 1. The addition is performed as follows:

$y = |x_1 + x_2|_m = \begin{cases} x_1 + x_2, & \text{if } x_1 + x_2 < m \\ x_1 + x_2 - m, & \text{if } x_1 + x_2 \geq m. \end{cases}$

Two n-bit adders are used; the first computes $x_1 + x_2$, while the second computes $x_1 + x_2 - m$. The carry bit generated from the second adder indicates whether or not $x_1 + x_2$ is greater than $m$. A multiplexer, controlled by the carry, selects the correct output. In the second method, a lookup table is used to correct the incorrect residue states $(2^n - m)$, Fig. 2. The first algorithm of modulo addition has a time complexity of $O(\log n)$, and the second algorithm is not suitable for medium and large moduli.
The idea of representing a number as a carry and a sum can be used in the modulo addition to obtain a scheme that has a constant speed which does not depend on the number of bits. The modulo adder is used to add two numbers $A$ and $B$ in modulo $m$. Fig. 3 shows that $A$ is represented as a pair of numbers $(A_s, A_c)$, $B$ is also represented as $(B_s, B_c)$, and the output $C$ is represented as $(C_s, C_c)$. Each number is represented as a group of sum bits and carry bits. There is no unique representation for $A_S$ and $A_C$. The condition that needs to be satisfied is 

$$ |A_s + A_c|_m = |A|_m. $$

One possible representation is 

$$ A_s = |A|_m, \quad A_c = 0. $$

The choice of a representation has no implication on the complexity of the design. With such representation, four numbers $(A_s, A_c, B_s, B_c)$ need to be added, and two steps of CSA are required. After the addition process we need to detect if $-M$ or $2^m - M$ is required to adjust the result. The adjusting process takes at most three steps. Since the adder has a fixed number of steps—five—no matter how long $A$ and $B$ are, it can be used in a multioperand pipelined addition scheme [14].

3.1. The Modulo Addition Algorithm

The proposed algorithm for modulo $m$ addition of two numbers can be described as follows.

**Algorithm modulo add $(A, B, \text{Result})$**

*Input:* Two variables $A$ and $B$ in modulo $m$, $A$ is represented as $A_s$ and $A_c$. $B$ is represented as $B_s$ and $B_c$. All variables are $n$ bit numbers ($2^{n-1} < m \leq 2^n$).

*Output:* Variable $\text{Result}$ represented as $\text{Result}_s$ and $\text{Result}_c$. The relation between $A$, $B$, and Result is: Result = $|A + B|_m$.

**Procedure:**

begin

Do in parallel

begin

Call $\text{Sum}(\text{temp}_1, A_s, A_c, B_s)$

Call $\text{Carry}(\text{temp}_2, A_s, A_c, B_s)$

end

Do in parallel

begin

Call $\text{Carry}(\text{temp}_3, \text{temp}_1, \text{temp}_2, B_c)$

Call $\text{Carry}(\text{temp}_4, \text{temp}_3, \text{temp}_2, B_c)$

end

end

Case $(\text{temp}_a[n + 1] - \text{temp}_4[n + 1])$ of

0: Do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

Case $(\text{temp}_a[n + 1])$ of

0: do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

Case $(\text{temp}_a[n + 1])$ of

0: do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

Case $(\text{temp}_a[n + 1])$ of

0: do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

Case $(\text{temp}_a[n + 1])$ of

0: do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

Case $(\text{temp}_a[n + 1])$ of

0: do in parallel

begin

Result$_s$ := temp$_3$

Result$_c$ := temp$_4$

end

1: Do in parallel

begin

Call $\text{Sum}(\text{temp}_3, \text{temp}_s, \text{temp}_a, (2^n - m))$

Call $\text{Carry}(\text{temp}_a, \text{temp}_s, \text{temp}_a, (2^n - m))$

end

end case

end.

**Sum** $(A, B, C, D)$$

begin

Do in parallel $(1 \leq i \leq n)$

$$ A[i] := (B[i] \land C[i]) \lor (B[i] \land D[i]) \lor (C[i] \land D[i]) $$

end

Call $(A, B, C, D)$$

begin

$A[1] := 0$

Do in parallel $(1 \leq i \leq n)$

$$ A[i + 1] := B[i] \oplus C[i] \oplus D[i] $$

end

An implementation of the algorithm is shown in Fig. 4.
Theorem 1: The modulo adder scheme for adding two n-bit numbers in modulo \( m \) has an asymptotic time complexity \( O(1) \).

Proof: To prove that the number of steps is constant (five) we need to prove that the last carry is equal to zero in five or less steps. Induction is used to prove the correctness of the theorem on the number of bits \( n \).

1) Basis step: for \( n = 0 \), it means that we do not add any numbers and in this case the required number of steps is zero.
2) Induction hypothesis: assume for a fixed arbitrary \( n \geq 0 \) that the maximum number of steps is five.
3) Induction step: for numbers with \( n + 1 \) bits let:

\[
\eta = \text{temp}_n [n + 1] + \text{temp}_n [n + 2].
\]

Then we have the following cases.
(a) \( \eta = 0 \): then the carry propagation stopped at bit \( n \), and it ends after five steps at most according to the induction hypothesis.
(b) \( \eta = 1 \): then the correction is \( 2^{n+1} - m \) in step 3. Since \( m > 2^n \), then \( 2^{n+1} - m < 2^n \), which means that \( (2^{n+1} - m) [n] = 0 \). The worst case we get to have \( \text{temp}_n [n + 1] \) and \( \text{temp}_n [n + 2] \) to be equal to one. This means that \( \text{temp}_n [n + 1] = 0 \) and \( \text{temp}_n [n + 2] = 1 \). In this case the correction is done in two steps (step 3 and step 4).
(c) \( \eta = 2 \): then the correction is \( 2^m [2^{n+1} - m] \) in step 3. The worst case we get to have \( \text{temp}_n [n + 1] \), \( \text{temp}_n [n + 2] \), and \( 2^m [2^{n+1} - m] \) to be equal to one. Then \( \text{temp}_n [n + 1] = 1 \), \( \text{temp}_n [n + 2] = 1 \), and \( 2^{n+1} - M = 0 \). At step 4 \( \text{temp}_n [n + 1] = 0 \) and \( \text{temp}_n [n + 2] = 1 \). At step 5 \( \text{temp}_n [n + 1] = 1 \) and \( \text{temp}_n [n + 2] = 0 \). In this case the correction is done in three steps (steps 3-5).

As an example, the modulo addition of \( A = 1272 \) and \( B = 450 \) for \( m = 2050 \) is shown in Fig. 5. There is no unique representation for \( A \) and \( B \). One valid representation is shown in this figure. The detailed modulo addition operation is shown in this example. In step 1 we get \( \text{temp}_{13} [3] = 1 \), and in step 2 we get \( \text{temp}_{13} [1] = 1 \), which means that at step 3 we have to add \( 2(2^n - M) \). At step 3 we get \( \text{temp}_{13} [3] = 1 \), which means that at step 4 we have to add \( 2^n - M \). At step 4 we get \( \text{temp}_{13} [3] = 0 \), which means that the addition process stops at step 4. The result of step 4 is the final result.

IV. MODULO ADDER EVALUATION

Using the VLSI model of computation for asymptotic complexity [15], a comparative study for the proposed adder is analyzed. For adder I (Fig. 1), using the binary adder of Brent and Kung [16], the complexity measures will be as follows:

\[
A = O(\log m \log \log m) = O(n \log n)
\]

\[
T = O(\log \log m) = O(\log n)
\]

\[
AT^2 = O(n(\log n)^2).
\]

For adder II (Fig. 2), using the complexity analysis of the correlation table of [17]:

\[
A = O(\log m \log \log m + m \log m) = O(n \log n + 2^n) = O(n 2^n)
\]

\[
T = O(\log \log m + \log m) = O(\log n + n) = O(n)
\]

\[
AT^2 = O(n 2^n).
\]

For the proposed adder,

\[
A = O(n)
\]

\[
T = O(1)
\]

\[
AT^2 = O(n).
\]

V. CONCLUSIONS

The modulo adder introduced in this paper has a total time-delay complexity of \( O(1) \) for adding two \( n \)-bit numbers in modulo \( m \). Based on the analysis of Section IV, this adder is the...
fastest and the most area efficient for large moduli. The proposed design has the following advantages.
1) It does not have any limitation on the size of the modulus.
2) It is quite modular, and it is a two-dimensional array of one type cell (full-adder).
3) It is easy to pipeline.
4) It is very efficient architecture for the implementation of the CRT decoding [14].

REFERENCES

Adjusting the Parameters in Elliptic-Function Filters
H. J. ORCHARD

Abstract—When designing elliptic-function filters there is usually some margin in performance to be distributed over the defining parameters. A recent paper offered some comparatively complicated formulas for use in this stage of the design. However, a simpler method, originally due to Darlington, is available and is described briefly.

Manuscript received April 28, 1989. This letter was recommended by Associate Editor T. R. Viswanathan. The author is with the Electrical Engineering Department, University of California Los Angeles, Los Angeles, CA 90024.

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, VOL. 37, NO. 5, MAY 1990
631

I. INTRODUCTION

The solution given by Vlcek and Unbehauen [1] to what they refer to as the "degree equation" for elliptic-function filters is exact, but rather complicated for the intended use in adjusting the parameters at the beginning of a design, and involves computing several elliptic functions of rational fractions of a quarterperiod. High precision is unnecessary at this stage in the design, and a much simpler formula which allows one to achieve the same end result with a pocket calculator was in fact given by Darlington [2] just 50 years ago, but seems to have been overlooked. The purpose of this note is to explain the formula in more detail than appeared in [2] and to expand on it slightly.

In the design of an elliptic-function filter one is given a specification for: The passband ripple, $a_p$, the minimum stopband loss, $a_s$, (both in decibels) and the elliptic modulus $k = \omega_p/\omega_s$, where $\omega_p$ is the passband-edge frequency and $\omega_s$ is the stopband-edge frequency. From these one must first find the smallest integral value of the degree $n$ that can be used. This choice for $n$ will normally cause the filter to be slightly better than is called for in the specification, so one can adjust the values of $a_p$, $a_s$, and $k$ that one uses in the design to allow some margin inside the specification. The relation between these four parameters, $a_p$, $a_s$, $k$, and $n$ is given by the "degree equation."

II. DEFINITION OF THE ELLIPTIC-FUNCTION FILTER

The power ratio for an elliptic-function filter is most conveniently defined by a pair of equations involving a parametric variable, exactly analogous to those for the well-known Chebyshev filter. The latter is defined by

$$10^{u/10} = 1 + e^{2\cos^2\mu u}$$

$$\mu = \cos u$$

where $u$ is the parametric variable and $n$ is the degree. The passband edge is normalized to $\mu = 1$ and the passband ripple is

$$a_p = 10\log(1 + e^2)$$

The equations for the elliptic-function filter are obtained by replacing the cosines in (1) by the Jacobian cd elliptic functions and take the form

$$10^{u/10} = 1 + e^{2\cd(\mu u K_1/\mu K_2)}$$

$$\mu = \cd(u; k)$$

As in (1), this definition holds for both odd and even values of the degree $n$, $K_1$ and $K$ are the real quarterperiods belonging to the elliptic functions with moduli $k_1$ and $k$ in (3a) and (3b) respectively. The frequency scale is still normalized to $\mu = 1$ at $\omega_p$, rather than to $\sqrt{\omega_p\omega_s}$ as in [1] and [2]; the latter normalization, though nicer for theoretical work, is a nuisance in practical design. The passband ripple is again given by (2).

In order for (3) to define a rational function, the period rectangle of $\cd(nu K_1/\mu K_2; k_1)$, in the $u$ plane, must fit exactly $n$ times into the period rectangle of $\cd(u; k)$, just as the period strip of $\cos nu$ fits $n$ times into the period strip of $\cos u$. This

$^1$The cd function is the same as the sn function by a real quarter period, just as the cosine is the same as the sine shifted by $\pi/2$. Using the cd function rather than the sn function as in [2], allows one to describe both odd and even degree cases with one common formula.