VLSI structures for bit-serial modular multiplication using basis conversion

M.G. Parker
M. Benaissa

Indexing terms: Mathematical techniques, bit-serial multipliers

Abstract: This paper proposes design techniques for the efficient VLSI implementation of bit-serial multiplication over a modulus. These techniques reduce multiplication into simple cyclic shifts, where the number representation of the data is chosen appropriately. This representation will, in general, be highly redundant, implying a relatively poor throughput for the multiplier. It is then shown how, by splitting the multiplier into two pipelined multipliers, the throughput of the unit can be increased, whilst still retaining a cyclic-shift implementation. The split multiplier requires a mid-computation basis conversion, and the two number representations, used within the unit, are only moderately redundant. Thus, high-throughput, bit-serial multipliers are achieved, with most of the complexity contained within systolic basis converter modules. The multipliers are applicable to the VLSI implementation of high-throughput, signal processing operations performed over finite fields, in particular, transform and filter operations.

1 Introduction

Recent advances in VLSI technology have suggested novel approaches to the implementation of arithmetic units over algebraic rings and fields other than real or complex [1–4]. These new approaches have been spurred on by the need for fault-tolerant, systolic architectures, where throughput is maximised and design complexity minimised. Residue number systems (RNSs) perform arithmetic over a modulus, M where M can be expressed, $M = \prod_{i=0}^{n-1} m_i$, and all arithmetic can be decomposed into a combination of smaller, parallel, arithmetic sub-computations, thus reducing the granular dimension of any consequent VLSI implementation [5, 6]. However, some or all of these m_i can still be quite large, hence the need for efficient implementations of modular arithmetic units [2, 4, 7, 8].

The concept of basis and basis flow (BF) are defined, and it is shown how, for a specified BF, multiplication by an element q, mod m, can be implemented using the exponent, e, as input, where $\langle g^e \rangle_m = q$, g is an element of the field/ring, and the basis $\alpha = g$. $\langle * \rangle_m$ means the residue of $*$, mod m. A decomposition of the original multiplication is developed into two pipelined multiplications, effectively replacing e with e_1 and e_2. A basis converter (BC) module is inserted between the two multipliers to facilitate the design of the second multiplier, and necessary criteria are developed for the chosen bases, to achieve a maximum throughput implementation. An example is given of a multiplier, firstly using a direct implementation, then, secondly, a split implementation, and a suitable application of the split multiplier to a number-theoretic transform (NTT) is discussed. These multipliers are particularly suitable for use within the word-serial implementation of general transforms over finite fields, where many products of a single element are required at any one time. Systolic architectures are presented for basis conversion, and the benefits in area achieved by a split-multiplier/systolic BC implementation are briefly assessed.

Paper 1527E (C2, E10), first received 15th July 1993 and in revised form 15th June 1994

The authors are with the School of Engineering, University of Huddersfield, Huddersfield, HD1 3DH, United Kingdom

2 Multiplication using exponents

We desire to implement the multiplication

$$r = \langle xq \rangle_m \tag{1}$$

where x, q and r are elements of the ring/field of integers $0, \ldots, m-1$, referred to in this paper as H_m, and x will be represented using the following BF parameters (see Appendix):

$$R = 2, \alpha, d, j, j' \quad \text{and} \quad Z$$

q will be represented by its exponent e, where $q = \langle g^e \rangle_m$ and g is an element of H_m. g can have one of the following two properties.

(i) $\langle g^{f_1} \rangle_m = \langle g^e \rangle_m$ where $f_1 > e$ and $\langle g^{f_2} \rangle_m \neq \langle g^e \rangle_m$ for any $e < f_2 < f_1$, in which case g is cyclic, mod m, of order $(f_1 - e)$

(ii) $\langle g^{f_3} \rangle_m = 0$ and $\langle g^{f_4} \rangle_m \neq 0$ for any $0 \leqslant f_4 < f_3$, in which case g is not cyclic.

In the first case, $(f_1 - e)$ unique values of $\langle g^e \rangle_m$ exist. In the second case, f_3 unique, non-zero values of $\langle g^e \rangle_m$ exist. These restrictions, in turn, limit the possible values of q which can be represented in this form. We observe that, if m is prime, g can be chosen to possess the first property, such that $O(g)_m = m - 1$, where $O(*)_m$ means the order of $*$, mod m. Thus, over a prime modulus, any value of q, from $0, \ldots, m-1$, can be represented as $\langle g^e \rangle_m$.

If we now define the BF so that $\alpha = g$, then eqn. 1 can be implemented by shifting x. This is simply an extension of the well known concept of shifting of binary data to perform power of 2 multiplications [9–11]. However, in general there will be a shift overflow problem. For

instance, if $m = 13$, and we define a BF with

$$R = 2,\ \alpha = 2,\ d = 1,\ j = j' = 4 \quad \text{and} \quad \boldsymbol{Z} = [1]$$

then eqn. 1 can be accomplished, where q ranges from $\langle g^0 \rangle_{13}$ to $\langle g^{11} \rangle_{13}$, by using 0 to 11 shifts of x, if g is chosen $= \alpha = 2$. Unfortunately, we then have to deal with multiple overflow into the fifth bit of the BF. If, instead, the BF of x has $j' = 12$ then eqn. 1 can be implemented using cyclic shifts. This simple solution is at the price of data wordlength, and, hence, throughput. Another possibility would be to define the BF of x as follows:

$$R = 2,\ \alpha = 2,\ d = 2,\ j = j' = 6,\ \boldsymbol{Z} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

This is only one of many possibilities.

Again, with this BF, eqn. 1 can be implemented using cyclic shifts between BF rows, at the price of throughput and area. This example demonstrates how the multiplicative implementation complexity may be avoided by a suitable BF for x, and a suitable exponent representation for q. By increasing the redundancy of the BF, even overflow circuitry may be eliminated. On the downside, we now need special exponent coding of q, an unusual redundant number representation (RNR) for x and a potentially decreased throughput capacity for bit-serial implementations, due to increased wordlength, especially for large m. In the following, $N = O(g)_m$, $O(*)_m$ means the order of $*$, mod m, and $\alpha = g$.

To ensure a cyclic shift implementation of eqn. 1, it is sufficient to consider the destination of the most significant digit (msd) column in the BF after shifting left by one column (multiplying by α). A cyclic shift is defined by the avoidance of additive combinations of overflow bits after shifting left, and is only achieved by passing the msd to the least significant digit (lsd) column of the representation, with or without BF row permutation. This is only possible if

$$W_s \alpha^{j_c} = W_s \alpha^{\langle j_c \rangle_N} = W_t$$

for

$$s, t \in [0, 1, \ldots, d-1]$$

and

$$j_c \in [j, j+1, \ldots, j'] \tag{2}$$

for each row, with the weights, W_i, as defined in Appendix 10.1. Note that $\alpha^{j_c} = \alpha^{\langle j_c \rangle_N}$ as $\alpha^N = 1$, mod M.

If cyclic shifts are performed without row permutation, $s = t$ and eqn. 2 becomes $\alpha^{j_c} = 1$, therefore $j_c = kN$ and

$$kN \in [j, j+1, \ldots, j'] \text{ for } k \text{ a positive integer} \tag{3}$$

which is clearly always appropriate for one row solutions, i.e. $d = 1$.

The maximum element order, mod prime m, is $N = m - 1$. For large, prime m, this N is much greater than the minimum j necessary to span H_m, j_{min}, for a given BF. For solutions without BF row permutation, expr. 3 implies that cyclic shifting is only possible if j' is much greater than j_{min}, making the number representation highly redundant, and reducing throughput drastically. The alternative using row permutation is given by eqn. 2, where an increase in d can lower the requirements on j' in spite of N. However, large d is not desirable from an implementation point of view, and the rest of this paper is concerned with solutions where d is kept small. With or without BF row permutation we conclude that, for a given d, j' will have to increase in proportion to N.

In the next Section, we show how j' can be lowered by reducing the effective N. This is achieved by splitting the multiplication and inserting an appropriate mid-computation basis conversion.

3 Lowering wordlength by using a split-multiplier

Let us consider eqn. 1 where m is large and prime. From expr. 3 we see that, for large N, j' must become much greater than j_{min} to ensure a cyclic shift implementation. Alternatively, from eqn. 2 d and/or j' may be increased, with the weights, W_i, suitably chosen. As mentioned previously, large d is not considered. Therefore j' becomes much greater than it's theoretical minimum necessary to span H_m. One method of lowering j' is to split eqn. 1 into two sub-computations:

$$r = \langle qx \rangle_m = \langle q_1 q_2 x \rangle_m = \langle q_1 y \rangle_m \tag{4}$$

where $y = \langle q_2 x \rangle_m$, $q_1 = \langle g_1^{e_1} \rangle_m$, $q_2 = \langle g_2^{e_2} \rangle_m$, $q = \langle g^e \rangle_m$, $g_1 = \langle g^{b_1} \rangle_m$, $g_2 = \langle g^{b_2} \rangle_m$ and e, e_1, e_2, b_1, b_2 are the minimum possible values necessary to satisfy the above congruences.

Hence, $g^e = g_1^{e_1} g_2^{e_2} = g^{e_1 b_1} g^{e_2 b_2}$, all mod m, and

$$e = \langle e_1 b_1 + e_2 b_2 \rangle_N \quad \text{where } N = O(g)_m \tag{5}$$

Thus, instead of performing the multiplication using e, we can perform it using two inputs, e_1, e_2, and, for each of the N unique values of e, there should be at least one pair of values e_1, e_2. This is only possible if the following condition is met:

Condition 1: gcd $(b_1, b_2) = 1$ where gcd $(*, *)$ is the greatest common denominator. If Condition I is met, then three options exist for the values of b_1 and b_2:

Option 1:

gcd $(b_i, N) = 1$ for $i \in [1, 2]$

Option 2:

gcd $(b_i, N) = 1$, gcd $(b_k, N) \neq 1$ for $i, k \in [1, 2]$, $i \neq k$

Option 3:

gcd $(b_i, N) \neq 1$ for $i \in [1, 2]$

We will consider the implementation of expr. 4 using each of these options, and the architectures are shown in Figs. 1, 2 and 3. For each case we assume an α-BF input

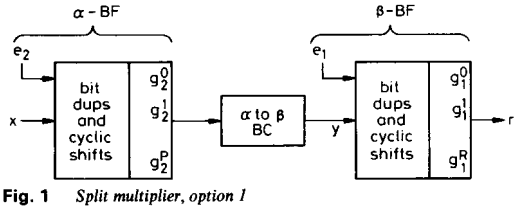

Fig. 1 *Split multiplier, option 1*
Where $P = (N \text{ DIV } b_2)$, $R = b_2 - 1$ or $P = b_2^{-1} - 1$, $R = (N \text{ DIV } b_2^{-1})$

to stage 1, where $g_2 = \alpha$, and a β-BF to stage 2, where $g_1 = \beta$. Hence, for all three options, an α to β basis converter (BC) is inserted between the two stages (see Section 6). We define $N_1 = O(g_1)_m$ and $N_2 = O(g_2)_m$.

3.1 Option 1

The schematic for this multiplier is shown in Fig. 1 and we have $N_1 = N_2 = N$. Without loss of generality, we assume that $g_1 = g$, therefore $b_1 = 1$ and eqn. 5 simplifies

to

$$e = \langle e_1 + e_2 b_2 \rangle_N$$

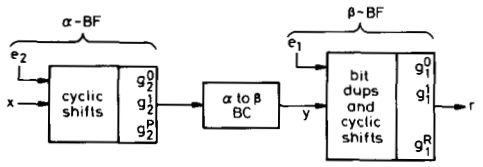

Fig. 2 *Split multiplier, option 2*
Where $P = N_2 - 1$, $R = b_2 - 1$

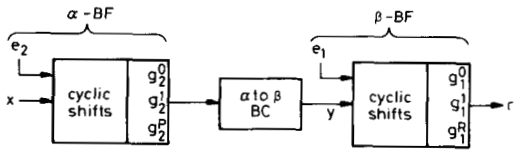

Fig. 3 *Split multiplier, option 3*
Where $P = N_2 - 1$, $R = N_1 - 1$

where

$$0 \leqslant e_1 < b_2, \quad 0 \leqslant e_2 \leqslant (N \text{ DIV } b_2)$$

or

$$0 \leqslant e_1 \leqslant (N \text{ DIV } b_2^{-1}), \quad 0 \leqslant e_2 < b_2^{-1}$$

and

$$\langle b_2 b_2^{-1} \rangle_N = 1 \tag{6}$$

Although the ranges for e_1 and e_2 are smaller than the range for e, the implementation of either stage using cyclic shifts is only possible if eqn. 2 is satisfied. As neither N_1 nor N_2 are less than N, the requirements on j' are not lowered. If, in spite of this, a lower j' is chosen to increase throughput, eqn. 2 is no longer satisfied, and the implementation of overflow is less straightforward. But the implementation of small power multiplications (SPMs), for g_1 or g_2, can, in fact, be simplified to become a single addition of cyclic shifted data, depending on the ranges of e_1 and e_2, and j' $(=j'_\alpha = j'_\beta)$. These solutions perform the SPM addition by doubling the BF parameter d by cycling and duplicating the bits inherent in the original word. Thus, no explicit addition is required, only bit duplication and cyclic shifting (BDCS). This is simply implemented using only delay elements, but at the price of a doubled d. If followed by a BC unit, the delayed addition can be incorporated into the basis conversion at minimal extra hardware cost.

3.2 Option 2

The schematic for this multiplier is shown in Fig. 2. Again, without loss of generality, we assume that $g_1 = g \neq g_2$. Therefore $b_1 = 1$, $N_1 = N > N_2$, and eqn. 5 simplifies to

$$e = \langle e_1 + e_2 b_2 \rangle_N$$

where

$$0 \leqslant e_1 < b_2, \quad 0 \leqslant e_2 < N_2 \tag{7}$$

The ranges of e_1 and e_2 are smaller than the range of e and the implementation of $\langle q_2 x \rangle_m$ using cyclic shifts is possible with a much lower requirement for j'_α, as N_2 can be substituted for N in exprs. 2 and 3.

However, for $\langle q_1 y \rangle_m$, the cyclic-shift condition must still use N in exprs. 2 or 3, and, as j' must remain constant throughout the system, we require j' = greater of (j'_α, j'_β) and the lower requirement for j'_α is nullified by the higher requirement for j'_β. This situation is avoided by noting that SPMs for $\langle q_1 y \rangle_m$ can, in fact, be implemented using bit duplication and cyclic shifting (BDCS), see option 1, without satisfying exprs. 2 or 3, thereby lowering the requirements for j'_β and j', and achieving a greater throughput rate, due to a reduced data wordlength. Note, this multiplier can equally well be reversed, where the BDCS block comes first, followed by the cyclic shifts, and this has the added advantage that the BDCS block is always followed by a BC, enabling, as with option 1, the delayed addition to be accomplished as part of the basis conversion.

3.3 Option 3

The schematic for this multiplier is shown in Fig. 3. We have $N_1, N_2 < N$, and for eqn. 5

$$0 \leqslant e_1 < N_1 \quad 0 \leqslant e_2 < N_2 \tag{8}$$

The ranges of e_1 and e_2 are smaller than the range of e, and, as N_1 and N_2 are substituted for N in exprs. 2 and 3, both multiplications can be implemented using cyclic shifts where the requirements on j' = greater of (j'_α, j'_β) are lower than the single multiplier case. As an aside, we note that this option satisfies, precisely, the conditions necessary for a two-stage, N-point, prime-factor DFT over H_m, where N_1, N_2 are the factored transform lengths [12].

For all three of the above options, suitable BF parameters and choices for g_1 and g_2 allow the lowering of the minimum j' required, for a given d. Obviously, the lowest possible j' is bounded by the minimum BF requirements necessary for a BF to span H_m. The throughput gain is at the cost of a BC module, inserted between multiplication stages, and a moderate increase in BF redundancy. Only option 3 implements the multiplications solely as cyclic shifts. The other two options require SPMs, implemented using cyclic shifting and implicit single additions, the additions being realised by increasing the redundancy of the BF.

3.4 Evaluation of e_1 and e_2

The multiplier scheme described above requires the input of two exponent indices, e_1 and e_2, which are derived from q. Although these indices could always be obtained using ROMs addressed by q, this option is not ideal especially for large moduli, m, where ROM size increases greatly. For general modular multiplication, the split multiplier would only become competitive when efficient means can be found to compute e_1 and e_2 given q. This is the subject of future research. A general modular multiplier scheme, again using basis conversion, is out-lined in Reference 13, but for this paper, the split multiplier is advocated primarily for functions that require fixed multiplication, such as for fixed filters or the NTT described later in this paper. For such tasks, e_1 and e_2 are implicitly embedded in the hardware structure, and their explicit computation is avoided.

The next Section presents an example, firstly, using a single-basis implementation and, secondly, a split implementation, using option 3, and compares the throughput requirements and area for the two methods.

4 Example implementation of multiplier

In this Section, an example is given for option 3 to demonstrate how splitting the multiplication of eqn. 1

can reduce j', and consequently increase throughput. We will operate over the field, H_m, where $m = 13$. From eqn. 1, we choose $q \in [0, 1, \ldots, m-1]$, i.e. all possible multiplications are required.

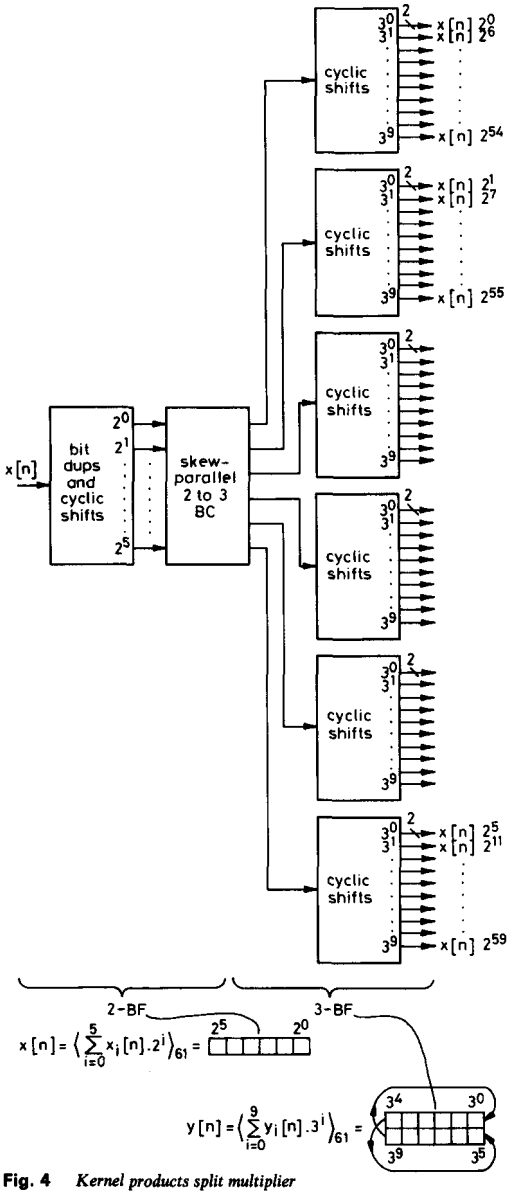

Fig. 4 *Kernel products split multiplier*
$g^N = 2^{60} = 1, g_1^{b1} = 2^6 = 3 = g_2$ and $g_2^{b2} = 3^{10} = 1$ (mod 61)

Firstly, we implement eqn. 1 directly then we consider a split implementation using option 3.

4.1 Single basis implementation

Let x have an input BF which is a standard 4-bit binary representation, as follows

$$R_\alpha = 2, \quad \alpha = 2, \quad d_\alpha = 1, \quad j_\alpha = 4,$$
$$j' \text{ (to be determined)}, \quad \boldsymbol{Z} = [1]$$

We note that this BF spans H_m, and is also a minimum basis with minimal redundancy, as $j_\alpha = j_{min}$ for the given BF.

As $O(\alpha)_m = 12$, the input BF is already suited to an implementation using shifts and we choose $g = \alpha = 2$. To implement eqn. 1 using cyclic shifts, eqn. 2 must be satisfied. For our chosen input BF, there is no BF row permutation and expr. 3 is given by

$$k12 \in [4, 5, \ldots, j'] \tag{9}$$

We see that the lowest j' necessary to satisfy expr. 9 is given by $j' = 12$. Hence, for this particular example, the multiplier can be implemented using cyclic shifts if at least $j' - j = 12 - 4 = 8$ delay cycles are introduced between each consecutive, 4-bit, input word.

4.2 Split implementation

Let us now implement eqn. 4. We choose $b_1 = 3$ and $b_2 = 4$, and note that

$$(b_1, b_2) = 1, \quad (b_1, N) = 3 \quad \text{and} \quad (b_2, N) = 4$$

These equations satisfy condition I and option 3, therefore suggesting a cyclic-shift implementation with a reduced j'. We note that $g_1 = \langle 2^3 \rangle_{13} = 8$ and $g_2 = \langle 2^4 \rangle_{13} = 3$. Eqn. 5 implies the following mapping from e to (e_1, e_2)

$$e = \langle 3e_1 + 4e_2 \rangle_{12} \tag{10}$$

where $0 \leqslant e_1 < N_1$, $0 \leqslant e_2 < N_2$ and $N_1 = O(g_1)_m = 4$, $N_2 = O(g_2)_m = 3$.

We choose an input α-BF for x, where $\alpha = g_2$, as follows

$$R_\alpha = 2, \quad \alpha = 3, \quad d_\alpha = 2, \quad j_\alpha = 3$$
$$j'_\alpha = j' \text{ (to be determined)}, \quad \boldsymbol{Z}_\alpha = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

We note that this α-BF is a minimum basis, i.e. $j_\alpha = j_{min}$ for the given BF, and contains a moderate amount of redundancy.

We require multiplications of x by powers of g_2 from g_2^0 to $g_2^{N_2-1}$ and we substitute N_2 for N into exprs. 2 and 3, giving

$$\alpha^{jc} = 1 \Rightarrow k3 \in [3, 4, \ldots, j'] \quad \text{for } s = t \tag{11}$$

and

$$\alpha^{jc} = 3^{jc} = -1 \quad \text{for } s \neq t \tag{12}$$

As $O(3)_{13}$ is odd, eqn. 12 has no solution and expr. 11 gives a lowest value of $j' = 3$. Therefore $y = \langle xq_2 \rangle_m$ can be implemented using cyclic shifts within each row, where $j'_\alpha = 3$.

We now convert from an α-BF to a β-BF where we choose the β-BF as follows

$$R_\beta = 2, \quad \beta = 8, \quad d_\beta = 2, \quad j_\beta = 3,$$
$$j' \text{ (to be determined)}, \quad \boldsymbol{Z}_\beta = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

We note that this is a minimum β-BF, and contains a moderate amount of redundancy.

We require multiplications of y by powers of g_1 from g_1^0 to $g_1^{N_1-1}$ and we substitute N_1 for N into exprs. 2 and 3, giving

$$k4 \in [3, 4, \ldots, j'] \quad \text{for } s = t \tag{13}$$

and

$$\beta^{jc} = 8^{jc} = -1 \quad \text{for } s \neq t \tag{14}$$

where $j_\beta\ (=3) \leqslant j_c \leqslant j'$. Expr. 13 gives a lowest value of $j' = 4$, and eqn. 14 gives a lowest value of $j' = 6$. Therefore $r = \langle yq_1 \rangle_m$ is best implemented using cyclic shifts within each row, where $j'_\beta = 4$.

The value of j' necessary to implement both multiplications as cyclic shifts is given by

$$j' = \text{greater of } (j'_\alpha, j'_\beta) = 4$$

Therefore $j' - j_\alpha = 4 - 3 = 1$ delay cycle is introduced between each consecutive input word.

Thus, by splitting the multiplication, the throughput rate has been increased from $j' = 12$ to $j' = 4$ and all multiplications can be implemented using cyclic shifts. The area cost is seen in terms of increased redundancy in the α-BF and β-BF number representations and the necessary insertion of an α to β BC.

5 Application of the split-multiplier to a bit-serial, word-serial number-theoretic transform

The number-theoretic transform (NTT) is a discrete Fourier transform (DFT) over a finite field, given by

$$X[k] = \left\langle \sum_{n=0}^{N-1} x[n] g^{nk} \right\rangle_m \quad 0 \leqslant n, k \leqslant N-1 \tag{15}$$

where $O(g)_m = N$, $\langle N^{-1} \rangle_m$ exists and is unique, and $g^p - 1$ is relatively prime to m for $1 \leqslant p \leqslant N$.

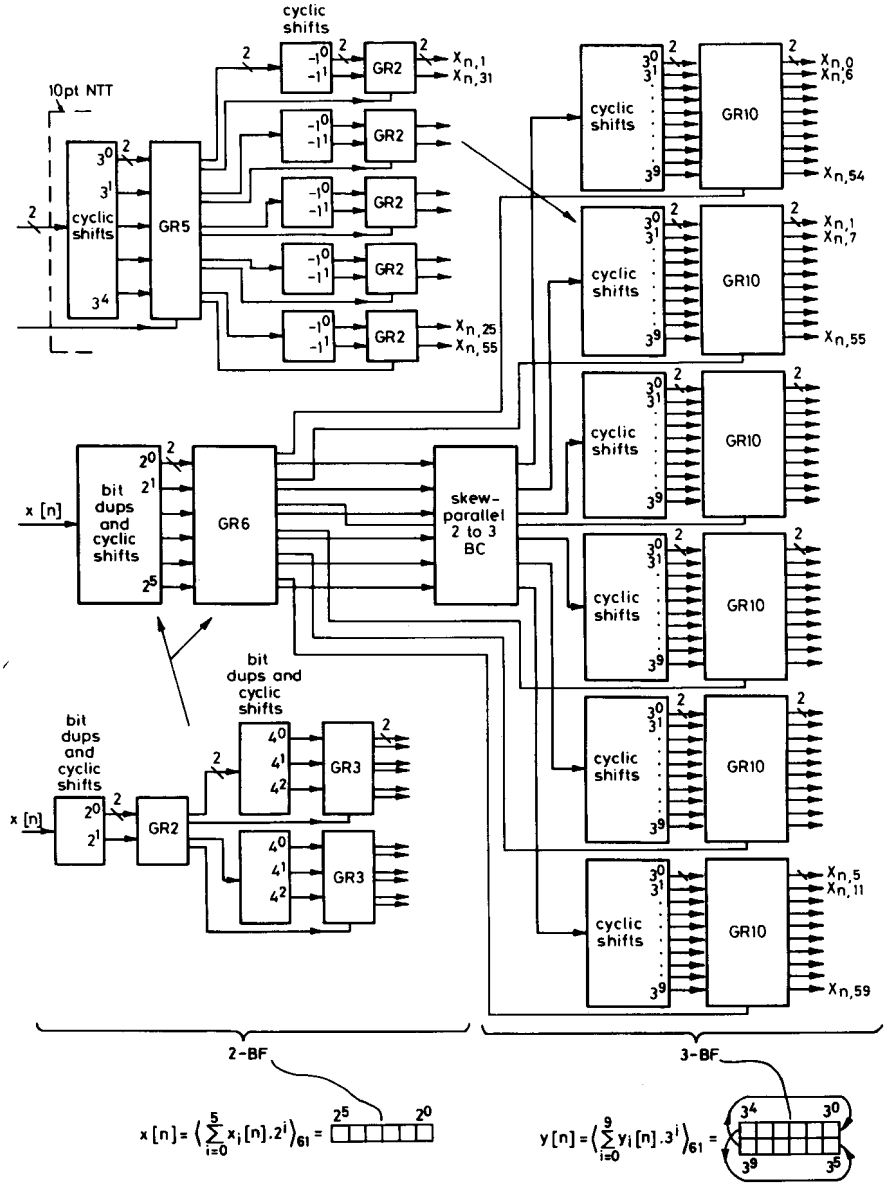

Fig. 5 *NTT product generator using a split-multiplier design*
$g^N = 2^{60} = 1$, $g_1^{b_1} = 2^6 = 3 = g_2$ and $g_2^{b_2} = 3^{10} = 1 \pmod{61}$

It can be cast as the summation of a series of products

$$X[k] = \left\langle \sum_{n=0}^{N-1} X_{nk} \right\rangle_m \quad 0 \leqslant n, k < N \tag{16}$$

where $X_{n,k} = \langle x[n]g^{nk} \rangle_m$.

The split multipliers, developed in this paper, can be used to compute $X_{n,k}$. We choose an input BF, where $\alpha = g$, and split g into g_1 and g_2. As the kernel multiplications within the NTT are fixed, we do not input e_1 or e_2 directly. Instead, the architecture inherently provides all possible multiples of $x[n]$, from $\langle x[n]g^0 \rangle_m$, ..., $\langle x[n]g^{N-1} \rangle_m$.

An example of this all encompassing multiplier architecture is shown in Fig. 4, for an input BF with the following parameters

$$R = 2,\ \alpha = 2,\ d_\alpha = 1,\ j_\alpha = 6,\ j'_\alpha = 6 \text{ and } \boldsymbol{Z}_\alpha = [1]$$

We consider multiplications of $x[n]$ by powers of $g = \alpha = 2$ from g^0, ..., $g^{N-1=59}$, mod 61. g is split into $g_1 = g = 2$ and $g_2 = \langle g^6 \rangle_{61} = 3$, where $O(g_1)^{61} = N_1 = 60$ and $O(g_2)_{61} = N_2 = 10$. We note that $b_1 = 1$ and $b_2 = 6$ so condition I of Section 3 is met, and the splitting satisfies option 2. Thus, we can perform the first multiplication stage using a BDCS block, and the second stage using cyclic shifts. In between the two stages we insert a 2 to 3 parallel BC (parallel to cope with each of the six power-of-two products resulting from the first stage). The second stage is performed using the following BF

$$R = 2,\ \beta = 3,\ d_\beta = 2,\ j_\beta = 5,\ j'_\beta = 6 \text{ and } \boldsymbol{Z}_\beta = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

The design is able to provide all possible kernel products without excessive area cost, because the complexity and cost is contained within the inter-stage BC. The kernel products come virtually free. The operation of the BDCS block is clarified in Appendix 10.2 and the parallel BC is explained in Section 6.

We now incorporate this comprehensive split multiplier within the NTT by appending, after each multiplier stage, a dynamic crossbar matrix, which selects the appropriate multiple of $x[n]$, for each k. This selection matrix will reconfigure for each successive $x[n]$ multiple, as appropriate to the NTT task. An example of this NTT product generation architecture is shown in Fig. 5, extending the example of Fig. 4, and the architecture performs the product generation for a 60pt NTT, mod 61. The dynamic crossbar matrices are labelled GR6, GR10, ... etc., where the number refers to the dimension of the matrix. For instance, a GR6 is a $6 * 6$ matrix. As is shown in Fig. 5, the sub-units can, themselves, have their multiplications split. $g_1 = 2$ can be split into $g_{1_1} = 2$ and $g_{1_2} = 4$, and $g_2 = 3$ can be split into $g_{2_1} = 3$ and $g_{2_2} = 3^5 = 60 = -1$. However, for this particular example, no further basis conversion is necessary as the α and β bases are already sufficiently convenient. This further splitting reduces the area requirements of the implementation, enhances localisation, and allows the NTT design to be constructed out of smaller NTT designs. (Note, the subsequent summation phase is dealt with separately, and is not considered here).

Thus, the split-multiplier design philosophy allows simple, high-throughput, low area NTT implementations, where the products are not computed explicitly, but implicitly, using basis conversion coupled with dynamic data routing. Further details of this form of NTT design can be found in References 14 and 15.

6 Basis conversion

Basis conversion can always be accomplished using ROMs. However, for large moduli, m, the size of look-up table increases dramatically, and more efficient conversion techniques need to be found. Parhami [16] has suggested non-modular, systolic radix converter cells. In this paper we propose a similar concatenation of systolic basis converter (BC) cells which achieve a maximum throughput. A BC cell is shown in Fig. 6, along with a

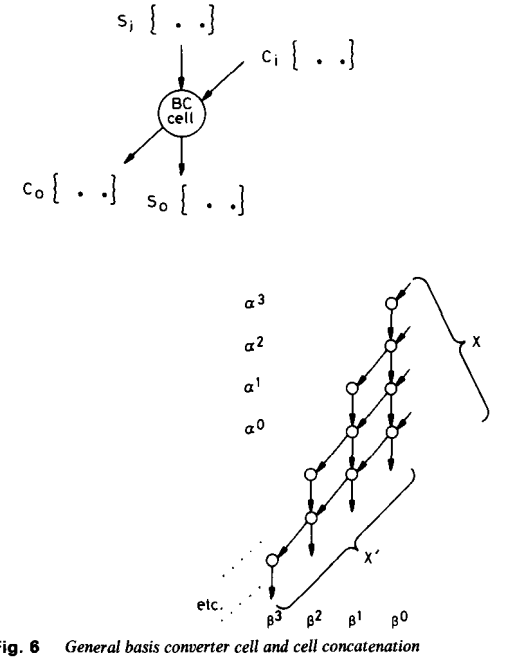

Fig. 6 *General basis converter cell and cell concatenation*

typical concatenation able to accept a skew-parallel data input. On each clock transition, data is transferred from input to output of each cell, governed by the equality

$$\alpha s_i + c_i = s_o + \beta c_o \mod m \tag{17}$$

where α and β are the input and output bases, respectively, and

$$\begin{aligned} &s_i \in \{\text{input-state integer set}\} \\ &c_i \in \{\text{input-carry integer set}\} \\ &s_o \in \{\text{output-state integer set}\} \\ &c_o \in \{\text{output-carry integer set}\} \end{aligned} \tag{18}$$

It is necessary for eqn. 17 to be satisfied for all possible values of s_i, c_i, s_o, c_o, and successful BC design will be achieved by minimising the number of output columns of Fig. 6 (output wordlength) for a given number of input diagonal rows (input wordlength). Note that the integer sets definition enables a level of abstraction independent of the target hardware. The concatenation scheme of Fig. 6 allows full pipelining at word and cell level to achieve maximum throughput digit-parallel solutions. Fully pipelined digit-serial solutions are then obtained by implementing only one diagonal row. To explain the operation of the BC, solutions will be proposed for the examples of Sections 4 and 5. In fact, a modification of eqn. 17 will often be required

$$\alpha s_i + c_i + k = s_o + \beta c_o \mod m \tag{19}$$

where k is an offset chosen to ensure satisfaction of eqn. 19 for all cases. This offset greatly widens the BC solution set, is implicitly implemented, i.e. no extra hardware, and the cumulative effect of each cell offset is observed in the final offset of the state output. By carefully choosing each cell offset, the cumulative offset can be made a multiple of the modulus, m, and is therefore eliminated from the final BC output.

Consider, first, the 2 to 3 BC, which may initially be required for the split-multiplier, mod 13, example of Section 4. Conventional binary input data will have a BF of

$$R_\gamma = 2, \quad \gamma = 2, \quad d_\gamma = 1, \quad j_\gamma = j'_\gamma = 4$$

and

$$\boldsymbol{Z}_\gamma = [1] \tag{20}$$

and it is desired to convert to a BF of

$$R_\alpha = 2, \quad \alpha = 3, \quad d_\alpha = 2, \quad j_\alpha = 3, \quad j'_\alpha = 4$$

and

$$\boldsymbol{Z}_\alpha = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{21}$$

Fig. 7 depicts a skew-digit-parallel 2 to 3 BC, defined by the cell parameters

Cell A: $s_i \in \{-\}$ $c_i \in \{0, 1\}$

$s_o \in \{0, 1\}$ $c_o \in \{-\}$

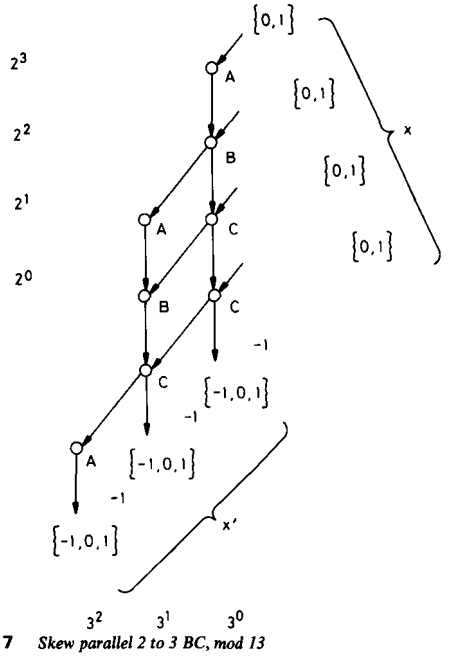

Fig. 7 *Skew parallel 2 to 3 BC, mod 13*

i.e. from (17), $2\{-\} + \{0, 1\} \in \{0, 1\} + 3\{-\}$

Cell B: $s_i \in \{0, 1\}$ $c_i \in \{0, 1\}$

$s_o \in \{0, 1, 2\}$ $c_o \in \{0, 1\}$

i.e. from (17), $2\{0, 1\} + \{0, 1\} \in \{0, 1, 2\} + 3\{0, 1\}$

$\Rightarrow \{0, 1, 2, 3\} \in \{0, 1, 2, 3, 4, 5\}$

Cell C: $s_i \in \{0, 1, 2\}$ $c_i \in \{0, 1\}$

$s_o \in \{0, 1, 2\}$ $c_o \in \{0, 1\}$

i.e. from (17), $2\{0, 1, 2\} + \{0, 1\} \in \{0, 1, 2\} + 3\{0, 1\}$

$\Rightarrow \{0, 1, 2, 3, 4, 5\} \in \{0, 1, 2, 3, 4, 5\}$

The input carried to the rightmost column, $\{0, 1\}$, matches the basis weights vector, $\boldsymbol{Z}_\gamma$, of eqns. 20. However, the output states of the bottom diagonal row, $\{0, 1, 2\}$, do not match $\boldsymbol{Z}_\alpha$ of eqn. 21. $\boldsymbol{Z}_\alpha$ represents the integers, $\{-1, 0, 1\}$. This is rectified by offsetting each output state, s_o, by -1 so that

$$\{0, 1, 2\} + -1 = \{-1, 0, 1\} \equiv \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

As shown in Fig. 7, the -1 offset is applied to each column, implying a cumulative offset of

$$-1 + 3(-1) + 3^2(-1) \mod 13 = 0$$

Therefore the offsets cancel and the output matches $\boldsymbol{Z}_\alpha$. d_γ and d_α of eqns. 20 and 21 imply a digit-serial BC. The skew parallel BC of Fig. 7 is easily serialised as all cells in a given column are of type C or subsets of C, (cells A and B), with input states compatible with their own output states. A serial version of the 2 to 3 BC is shown in Fig. 8,

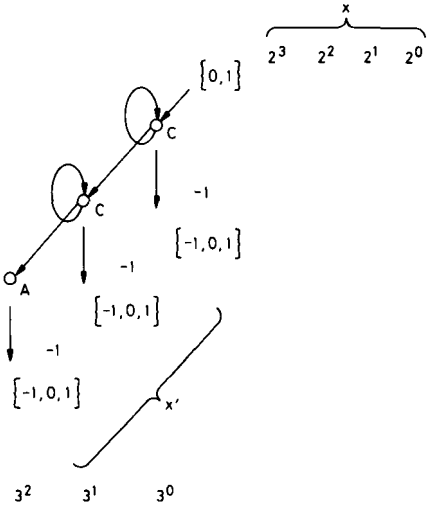

Fig. 8 *Serial 2 to 3 BC, mod 13*

where the intra-cell feedback is broken upon receipt of the final bit of each input word. Note, there is no inter-word input delay.

Consider, now, the realisation of a 3 to 8 BC. A skew parallel design is shown in Fig. 9 and is designed to match the input and output BFs, as given by eqns. 21 and

$$R_\beta = 2, \quad \beta = 8, \quad d_\beta = 2, \quad j_\beta = 4, \quad j'_\beta = 4$$

and

$$\boldsymbol{Z}_\beta = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{22}$$

The cell parameters are

Cell A: $s_i \in \{-\}$ $c_i \in \{-1, 0, 1\}$

$s_o \in \{-1, 0, 1\}$ $c_o \in \{-\}$

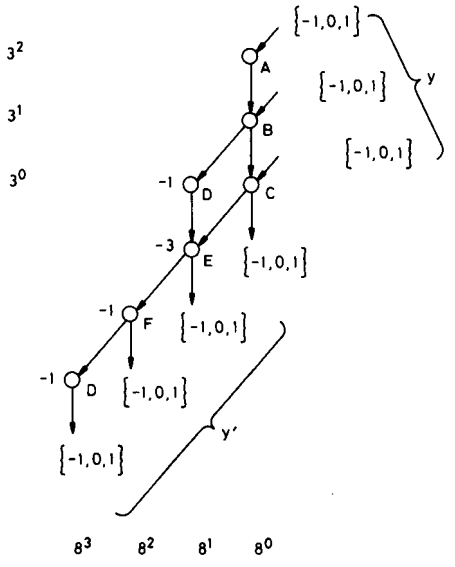

Fig. 9 *Skew parallel 3 to 8 BC, mod 13*

i.e. from (17), $3\{-\} + \{-1, 0, 1\} \in \{-1, 0, 1\} + 3\{-\}$

Cell B: $s_i \in \{-1, 0, 1\}$ $c_i \in \{-1, 0, 1\}$

$s_o \in \{-6, -5, -4, -1, 0, 1\}$ $c_o \in \{0, 2\}$

i.e. from (17), $3\{-1, 0, 1\} + \{-1, 0, 1\} \in \{-6, -5, -4, -1, 0, 1\} + 8\{0, 2\} \Rightarrow \{9, 10, 11, 12, 0, 1, 2, 3, 4\} \in \{7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4\}$

Cell C: $s_i \in \{-6, -5, -4, -1, 0, 1\}$ $c_i \in \{-1, 0, 1\}$

$s_o \in \{-1, 0, 1\}$ $c_o \in \{0, 1, 2, 3\}$

i.e. from (17), $3\{-6, -5, -4, -1, 0, 1\} + \{-1, 0, 1\} \in \{-1, 0, 1\} + 8\{0, 1, 2, 3\} \Rightarrow \{7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4\} \in \{7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4\}$

Cell D: $s_i \in \{-\}$ $c_i \in \{0, 2\}$

Offset, $k = -1$ $s_o \in \{-1, 0, 1\}$ $c_o \in \{-\}$

i.e. from (19), $3\{-\} + \{0, 2\} - 1 \in \{-1, 0, 1\} + 8\{-\}$

Cell E: $s_i \in \{-1, 0, 1\}$ $c_i \in \{0, 1, 2, 3\}$

Offset, $k = -3$ $s_o \in \{-1, 0, 1\}$ $c_o \in \{0, 1, 2, 3\}$

i.e. from (19), $3\{-1, 0, 1\} + \{0, 1, 2, 3\} - 3 \in \{-1, 0, 1\} + 8\{0, 1, 2, 3\} \Rightarrow \{10, 11, 12, 0, 1, 2, 3, 4, 5, 6\} - 3 \in \{7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4\}$

Cell F: $s_i \in \{-\}$ $c_i \in \{0, 1, 2, 3\}$

Offset, $k = -1$ $s_o \in \{-1, 0, 1\}$ $c_o \in \{0, 2\}$

i.e. from (19), $3\{-\} + \{0, 1, 2, 3\} - 1 \in \{-1, 0, 1\} + 8\{0, 2\} \Rightarrow \{0, 1, 2, 3\} - 1 \in \{-1, 0, 1, 2, 3, 4\}$

The cumulative offsets sum as follows

$$8 + 1 - (3.8) - (1.3.8) \mod 13 = 0$$

so offset is eliminated.

The skew parallel form of Fig. 9 is not easily serialised due to the non-identical nature of the cells in each column. However a single skew parallel 3 to 8 BC can be used to convert up to four power-of-3 multiples of the input without limiting throughput. The inter-word delay of 4 clock cycles ensures up to 4 BC conversions for every word input using the skew parallel form. As a serial form is not easily found, the BC is best suited to the combined realisation of up to four independent bit-serial split multipliers, mod 13, or a bit-serial 3-point or 12-point NTT, mod 13, or some similar task, where the BCs of Figs. 8 and 9 are interleaved with cyclic shifters (Fig. 3). Alternatively, the BCs of Figs. 7 and 9 may be combined to achieve a bit-parallel split-multiplier implementation. Thus, although the emphasis in this paper is on bit-serial solutions, skew parallel BCs can also be combined with cyclic shifters to form competitive bit parallel solutions.

Finally, a 2 to 3 BC, suitable for inclusion within the NTT example of Section 5 and Fig. 5, is shown in skew parallel form in Fig. 10. The cell parameters are

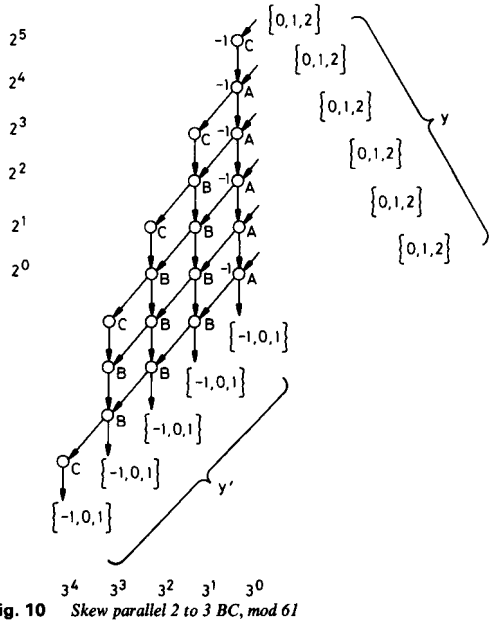

Fig. 10 *Skew parallel 2 to 3 BC, mod 61*

Cell A: $s_i = s_o \in \{-1, 0, 1\}$ $c_i \in \{0, 1, 2\}$

$c_o \in \{-1, 0, 1\}$

Cell B: $s_i = s_o \in \{-1, 0, 1\}$ $c_i = c_o \in \{-1, 0, 1\}$

Cell C: $s_i \in \{-\}$ $c_i \in \{-1, 0, 1\}$

$s_o \in \{-1, 0, 1\}$ $c_o \in \{-\}$

Note that cell A is adapted to accept two separate conventional binary bit streams

$$\boldsymbol{Z} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \equiv \{0, 1, 2\}$$

and therefore incorporates the BDCS implementation of power-of-2 multiplication, as explained in Appendix 10.2.

An important distinction between the BCs of Figs. 7, 8 and 10, and the BC of Fig. 9 is that the former are modulus independent BCs, incorporating only a few cell types, and are readily expanded to higher moduli. On the

other hand, the BC of Fig. 9 is modulus dependent, is much less regular, and is suited to only one modulus. Modulus independent BCs will be characterised by input-carry and output-state integer sets which span α and β respectively, for an α to β BC. For instance, an example of a modulus independent 11 to 7 BC could use the cell type specified by

$$\text{11 to 7 BC Cell:} \quad s_i = s_o \in \{0, 1, 2, 3, 4, 5, 6\}$$
$$c_i = c_o \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

The modulus dependent BCs do not satisfy these criteria and rely on offset modification for a localised interconnection scheme. As modulus size increases, the modulus-dependent solutions become increasingly difficult to design.

7 Area assessment of split-multipliers using systolic basis converters

By assuming a ROM implementaton for each BC cell, one can arrive at a figure for BC area cost for each of the multiplier examples discussed. The split-multiplier area cost can then be approximated to by its BC area cost, ignoring the cost for cyclic shifts, and this figure can be compared to a brute force implementation of fixed multiplication using a single ROM. A further comparison with the figure for N simultaneous fixed multiplications using N ROMs will then be quoted, to emphasise the suitability of the split multiplier for simultaneous product-generation tasks, such as the NTT. Note, for N simultaneous multiplications, the BC figure remains unchanged.

It should be remembered that, unlike the split multiplier, the ROM approach is essentially parallel, although not easily pipelined, especially for large moduli. From Table 1, the split-multiplier/systolic BC method would seem particularly suited to the case where many fixed multiplications are required simultaneously, such as the NTT. It is debatable whether the method is worth pursuing for moduli as small as 13, but for higher moduli the BC cell dimensions can become very small in comparison to the dimensions of the alternative ROM solutions. In summary, the method achieves a relatively low area by computing many products simultaneously, whilst also attaining a high throughput by matching element orders to the data wordlength.

Table 1: Comparative area assessment for split-multipliers using systolic basis converters

Multiplier, mod 13	Area cost (bits)
serial 2 to 3 BC	48
skew parallel 3 to 8 BC	154
total	202
single ROM (one fixed mult)	64
12 ROMs (twelve fixed mults)	768
Multiplier, mod 61	**Area cost (bits)**
skew-parallel 2 to 3 BC	504
total	504
single ROM (one fixed mult)	384
60 ROMs (sixty fixed mults)	23040

8 Conclusion

In this paper we have considered the implementation of bit-serial multiplication over a modulus, using number representations with increased redundancy. It is shown that a reduction in throughput rate, i.e. increased word-length, enables a simple multiplier implementation using only cyclic shifts. To improve on this throughput rate, whilst keeping the simplicity, the multiplier is split into two multipliers, and, for suitable choices of roots of the ring/field, the multiplier can be implemented using two pipelined cyclic shifts of smaller orders, thereby increasing the possible throughput rate. It is demonstrated how this split-multiplier design can be incorporated within an NTT design, enhancing its modularity and throughput, whilst minimising its implementation complexity and area. These benefits are made possible by the insertion of basis converter(s) (BC). Systolic BCs are briefly presented, based on a simple cell rule, and BC solutions are given for the multiplier examples presented earlier in the paper. It is clear that this technique for bit-serial multiplication can be generalised to more than two subcomputations, for use within a larger ring/field, and the multipliers are of particular use within the implementation of RNS-based systems, cryptosystems, signal processing operations such as transforms over a finite field, [3, 14, 15, 17], and error-correction systems. The design rules, developed herein, form a foundation for the synthesis of high-throughput, VLSI circuits for finite field operations.

9 References

1 McCLELLAN, J.H., and RADER, C.M.: 'Number theory in digital signal processing' (Prentice Hall, 1979)
2 JULLIEN, G.A.: 'Implementation of multiplication, modulo a prime number, with applications to number theoretic transforms', *IEEE Trans. Comput.*, 1980, **C-29**, (10), pp. 899–905
3 JULLIEN, G.A., BIRD, P.D., CARR, J.T., TAHERI, M., and MILLER, W.C.: 'An efficient bit-level systolic cell design for finite ring digital signal processing applications', *J. VLSI Signal Process.*, 1989, **1**, (3), pp. 189–208
4 SKAVANTZOS, A.: 'New multipliers modulo $2^N - 1$', *IEEE Trans. Comput.*, 1992, **41**, (8), pp. 957–961
5 TAYLOR, F.J.: 'Residue arithmetic: a tutorial with examples', *Computer*, 1983, **17**, (5), pp. 50–61
6 SODERSTRAND, M.A., JENKINS, W.K., JULLIEN, G.A., and TAYLOR, F.J.: 'Residue number system arithmetic: modern applications in digital signal processing' (IEEE Press, New York, 1986)
7 BAKER, P.W.: 'Fast computation of an $A * B$ modulo N', *Electron. Lett.*, 1987, **23**, (15), pp. 794–795
8 TAKAGI, N., and YAJIMA, S.: 'Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem', *IEEE Trans. Comput.*, 1992, **41**, (7), pp. 887–891
9 BENAISSA, M., PAJAYAKRIT, A., DLAY, S.S., and HOLT, A.G.J.: 'VLSI design for diminished-1 multiplication of integers modulo a fermat number', *IEE Proc. E*, 1988, **135**, (3), pp. 161–164
10 BALLA, P.C., and ANTONIOU, A.: 'Number-theoretic transform based on ternary arithmetic and its application to cyclic convolution', *IEEE Trans. Circuits Syst.*, 1983, **30**, (7), pp. 504–505
11 HONDA, M., KAMEYAMA, M., and HIGUCHI, T.: 'Residue arithmetic based multiple-valued VLSI image processor'. Proc. of 22nd Int. Symp. on *Multivalued Logic*, 1992, pp. 330–336
12 TRUONG, T.K., REED, I.S., HSU, I.S., SHYU, H.C., and SHAO, H.M.: 'A pipeline design of a fast prime factor DFT on a finite field', *IEEE Trans. Comput.*, 1988, **37**, (3), pp. 266–273
13 PARKER, M.G., and BENAISSA, M.: 'Using redundant number representations for efficient VLSI implementation of modular arithmetic'. Proc. of IEE Coll. on *Synthesis and Optimisation of Logic Systems*, Savoy Place, London, March 1994
14 PARKER, M.G., and BENAISSA, M.: 'A bit-serial, VLSI implementation of a 60-point NTT using binary and ternary bases'. International Conference on DSP for Communications, Warwick, U.K., October 1992
15 PARKER, M.G., and BENAISSA, M.: 'Bit-serial, VLSI architecture for the implementation of maximum-length number-theoretic transforms using mixed basis representations'. Proc. of ICASSP '93, Minneapolis, USA, 1993, **I**, pp. 341–344
16 PARHAMI, B.: 'Systolic number radix converters', *Comput. J.*, 1992, **35**, (4), pp. 405–409
17 POLLARD, J.M.: 'The fast Fourier transform in a finite field', *Math. Comput.*, 1971, **25**, (114), pp. 524–547

10 Appendix

10.1 Basis and basis flow (BF)

Basis refers to the weightings given to a group digits which, when evaluated and summed, represent a number. An α-basis means all weightings are successive powers of α (sometimes referred to as a canonical or primal basis). If $R = 2$, each digit is a binary digit, bit, having two values, 0, 1, although the concept can be extended to any digit range.

Basis flow (BF) refers to the flow characteristics of the basis data and the weighting pattern is shown below for one word, as a $j' * d$ vector,

$$\begin{array}{cccccccc} 0 & \cdots & 0 & W_0\alpha^{j-1} & W_0\alpha^{j-2} & \cdots & W_0\alpha^0 \\ 0 & \cdots & 0 & W_1\alpha^{j-1} & W_1\alpha^{j-2} & \cdots & W_1\alpha^0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & W_{d-1}\alpha^{j-1} & W_{d-1}\alpha^{j-2} & \cdots & W_{d-1}\alpha^0 \\ & & & \leftarrow \quad j \text{ clock cycles} \quad \rightarrow & & & \\ \leftarrow & & & j' \text{ clock cycles} & & \rightarrow & \end{array}$$

with

$$Z = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_{d-1} \end{bmatrix}$$

where the vector is assumed to be travelling right to left. j' is the word period and $j' - j$ is the inter-word delay, both in clock cycles, and d is the data path width. With no delay between successive words, $j' = j$ and, when $d = 1$, the data moves in a digit-serial fashion. Z is the basis weights vector, and $W_0, \ldots, W_{d-1}$ are the weightings associated with each data line. In the text, the BF will sometimes be loosely referred to as an α-BF. Each word using the above BF will equate to

$$W_0 \sum_{i=0}^{j-1} \alpha^i v_{i,0} + W_1 \sum_{i=0}^{j-1} \alpha^i v_{i,1}, \ldots, + W_{d-1} \sum_{i=0}^{j-1} \alpha^i v_{i,d-1}$$
$$= \sum_{k=0}^{d-1} W_k \sum_{i=0}^{j-1} \alpha^i v_{i,k}$$

where $v_{i,k}$ is the value (state) of each digit, and $0 \leqslant v_{i,k} < R$.

For a BF over a modulus M to span M, there must be at least one representation of each of $0, \ldots, M-1$ within the basis. If there is only one representation for each of $0, \ldots, M-1$ then the basis is non-redundant, otherwise it is redundant. j_{min} is the minimum value of j, for a given BF, necessary for the BF to span the modulus.

As an example of a BF, consider the following BF parameters,

$$R = 2, \ \alpha = 5, \ d = 2, \ j = 4, \ j' = 6 \quad \text{and} \quad Z = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Let us consider the i/p of the following $R = 2$, binary digit, data word, using a $d = 2$-bit-wide data bus

$$\begin{array}{cc} & 0\ 0\ 1\ 0\ 1\ 0 \\ \leftarrow & 0\ 0\ 0\ 1\ 1\ 0 \end{array}$$

If this is interpreted using the above BF, it evaluates to

$$1(\alpha^3 + \alpha^1) + 2(\alpha^2 + \alpha^1) = 130 + 60 = 190$$

Over a suitable modulus, m, this BF will span the range $0, \ldots, m-1$. For instance, if $m = 29$, the given BF spans m, where $j_{min} = 4$, and the above example equates to $\langle 190 \rangle_{29} = 16$.

The throughput for this example BF will be limited by j'. In other words, one data word is passed every 6 clock cycles.

10.2 Example BDCS block (see Section 5),

To clarify the function of the BDCS block, for the NTT split-multiplier example of Section 5, we present the format of the small power of 2 multiplications (SPMs), mod 61, as performed within the BDCS block,

If $x[n]$ is defined as $x_5 2^5 + x_4 2^4 + \cdots + x_0 2^0$ and represented by $x_5, x_4, x_3, x_2, x_1, x_0$ we can define the following

$$2^0 x[n] = x[n]$$
$$2^1 x[n] = x_4, x_3, x_2, x_1, x_0, x_5 + 0, 0, 0, 0, x_5, 0$$
$$2^2 x[n] = x_3, x_2, x_1, x_0, x_5, x_4 + 0, 0, 0, x_5, x_4, 0$$
$$2^3 x[n] = x_2, x_1, x_0, x_5, x_4, x_3 + 0, 0, x_5, x_4, x_3, 0$$
$$2^4 x[n] = x_1, x_0, x_5, x_4, x_3, x_2 + 0, x_5, x_4, x_3, x_2, 0$$
$$2^5 x[n] = x_0, x_5, x_4, x_3, x_2, x_1 + x_5, x_4, x_3, x_2, x_1, 0$$

(all operations mod 61)

and, instead of explicitly adding the shifted data, we can pass each output out of the BDCS block as a $6 * 2$-bit data stream, (i.e. $d = 2, j = j' = 6$).