Journal of VLSI Signal Processing, 3, 215–223 (1991)

Bit-Level Systolic Arrays for Modular Multiplication

ÇETIN K. KOÇ AND CHING YU HUNG
Department of Electrical Engineering, University of Houston, Houston, TX 77204

Received October 30, 1990, Revised January 15, 1991

Abstract. This paper presents bit-level cellular arrays implementing Blakley's algorithm for multiplication of two n-bit integers modulo another n-bit integer. The semi-systolic version uses $3n(n + 3)$ single-bit carry save adders and $2n$ copies of 3-bit carry look-ahead logic, and computes a pair of binary numbers (C, S) in $3n$ clock cycles such that $C + S \in [0, 2N)$. The carry look-ahead logic is used to estimate the sign of the partial product, which is needed during the reduction process. The final result in the correct range $[0, N)$ can easily be obtained by computing $C + S$ and $C + S - N$, and selecting the latter if it is positive; otherwise, the former is selected. We construct a localized process dependence graph of this algorithm, and introduce a systolic array containing $3nw$ simple adder cells. The latency of the systolic array is $6n + w - 2$, where $w = \lceil n/2 \rceil$. The systolic version does not require broadcast and can be used to efficiently compute several modular multiplications in a pipelined fashion, producing a result in every clock cycle.

Key Words: modular multiplication, carry save adders, sign estimation, systolic array, scheduling.

1. Introduction

Realization of some public key cryptosystems and authentication schemes using number-theoretic concepts, e.g., the RSA algorithm [1], and interactive computation protocols based on the quadratic residue problem [2], [3] requires fast computation of modular multiplication, i.e., the computation of

$$P = AB \pmod{N}, \tag{1}$$

for large positive integers A, B, and N. The security of the system is ensured by selecting operands having more than 200 decimal digits (664 bits). This requirement prohibits the use of table look-up techniques developed for the implementation of digital signal processing algorithms employing residue arithmetic [4]. Additionally, implementation of the cryptographic systems requires that the above computation be performed for several different values of N, whereas the classical applications of residue number systems use a fixed moduli set.

The computation of (1) can be achieved by first forming the product AB and then reducing it modulo N with a division. This method requires multiplication of two n-bit integers and division of a $2n$-bit integer by an n-bit integer. There exists a large volume of fast algorithms and hardware structures for binary multiplication and division [5], [6], which could be utilized for computing (1). However, since we are not interested in the quotient, the reduction process is actually simpler than a division. For example, Simmons and Tavares describe an algorithm and its NMOS implementation, which subtracts left-shifted versions of the modulus from the product in order to speed up the reduction process [7].

The shift-add steps of the multiplication process can also be interleaved with modular reduction [8], [9], [10]. This technique computes the product in n steps, where at each step one left shift, one addition, and at most two subtractions are performed.

The authors of this paper have proposed an algorithm in [11] for multi-operand modular addition

$$S = \sum_{i=1}^{k} X_i \pmod{N},$$

given integers N and $X_i < N$ for $1 \le i \le k$. The algorithm is based on Blakley's algorithm, and uses carry save addition and a novel technique for sign estimation, which is required during the reduction process. The modular reduction part of the algorithm is applied to modular multiplication to produce a word-serial, bit-parallel modular multiplier array [12].

In this paper, we review the use of carry save adders and give a rigorous analysis of the sign estimation technique. We then describe a semi-systolic multiplier array for modular multiplication which is a modified version of the one in [12]. The semi-systolic array

contains $3n$ rows of $n + 3$ single-bit carry save adder cells and $2n$ copies of 3-bit carry look-ahead logic. The array accepts the binary numbers A, B, and $-N$ as the input. After $3n$ clock cycles, a pair of integers (C, S) is produced such that $C + S \in [0, 2N)$. This result can be reduced to the correct range $[0, N)$ by computing $C + S$ and $C + S - N$ using carry propagate adders, and selecting the latter if it is positive; otherwise, the former is selected. The latency of the array is equal to $3n$, however, when several modular multiplications are computed in a pipelined fashion, the data rate is equal to the clock rate, i.e., a result is obtained at every clock cycle.

The semi-systolic array requires broadcasting the bits of A and the estimated sign bit across $n + 3$ cells; this would increase the duration of the clock due to signal propagation, and thus, decrease throughput rate. In this paper, we introduce a bit-level systolic array which requires only local communication, and thus, is more suitable for implementation in VLSI. The systolic modular multiplier is derived from the semi-systolic array by first constructing a localized process dependence directed acyclic graph (dag), and then embedding this dag in spacetime in order to obtain a systolic schedule. The resulting systolic array contains $3nw$ adder cells, and has a latency of $6n + w - 2$, where $w = \lceil n/2 \rceil$.

2. Carry Save Adders and Sign Estimation Technique

Blakley's algorithm for the computation of (1) is based on the application of Horner's method. In the following A_i denotes the ith bit of A, and n is the number of bits of modulus N, i.e., $n = \lfloor \log_2 N \rfloor + 1$.

Algorithm A
1. Set $P^{(0)} := 0$.
2. Repeat 2a and 2b for $i = 1, 2, 3, \ldots, n$.
   2a. $P^{(i)} := 2P^{(i-1)} + A_{n-i}B$.
   2b. $P^{(i)} := P^{(i)} \pmod N$.
3. End.

In Step 2a, we perform a left shift on the previous partial product $P^{(i-1)}$, and add the value $A_{n-i}B$ to obtain $P^{(i)}$. Then in Step 2b, $P^{(i)}$ is reduced to the range $[0, N)$. The final value of the partial product is equal to $P^{(n)} = P = AB \pmod N$. After the ith iteration of Step 2, we obtain $0 \le P^{(i)} < N$, and thus, in the following step we have $0 \le P^{(i+1)} < 3N$. This implies that at most two subtractions have to be performed to reduce the partial product to the desired range $[0, N)$.

Steps 2a and 2b of the algorithm require addition of n-bit numbers, which takes $O(n)$ time (gate delays) for the propagation of the carry from the least significant bit to the most significant bit. This delay should be avoided when n is large.

We propose the use of the word-serial, bit-parallel carry save adder (one-level CSA) [5] which, in one clock cycle, produces two n-bit numbers C and S from three n-bit numbers X, Y, and Z, such that $C + S = X + Y + Z$. The addition and subtraction in Algorithm A can be implemented using carry save adders, where the pair (C, S) represents the partial product $P = C + S$. The modular reduction operation in Step 2b requires a comparison operation and then a subtraction. Equivalently, we may compute the difference $\hat{P} = P - N$ first and then set P to $\hat{P}$ if $\hat{P} \ge 0$. The subtraction can be computed in one clock cycle using a carry save adder. However, testing the sign of $\hat{P}$, which is in the form of a carry-sum pair, requires the addition of two n-bit numbers (carry and sum), and thus takes $O(n)$ gate delays when a carry propagate adder is used. In the following, we introduce a technique which requires only $O(1)$ gate delays to estimate the sign of the partial product represented by a carry-sum pair.

If the exact sign of $C + S$ is computed, we can obtain the result in the correct range $[0, N)$. If an estimation of the sign is used, then we will prove that the range of the result becomes $[0, N + \Delta)$, where Δ depends on the precision of the estimation. Furthermore, since the sign is used to decide whether some multiple of N should be subtracted from the partial product, an error in the decision causes only an error of a multiple of N in the partial product, which is corrected later.

We define function $T(X)$ on an n-bit integer X as

$$T(X) = X - (X \bmod 2^t), \tag{2}$$

where $0 \le t \le n - 1$. In other words, T replaces the first least significant t bits of X with t zeros. This implies

$$T(X) \le X < T(X) + 2^t. \tag{3}$$

We reduce the pair (C, S) by performing the following operation Q times:

A. $(\hat{C}, \hat{S}) := C + S - N$.
B. If $T(\hat{C}) + T(\hat{S}) \ge 0$ then set $C := \hat{C}$ and $S := \hat{S}$.

In Step B, the computation of the sign bit R of $T(\hat{C}) + T(\hat{S})$ involves $n - t$ most significant bits of $\hat{C}$ and $\hat{S}$. The above procedure reduces a carry-sum pair from the range

$$0 \le C_0 + S_0 < (Q + 1)N + 2^t$$

to the range

$$0 \le C_R + S_R < N + 2^t,$$

where (C_0, S_0) and (C_R, S_R) respectively denote the initial and the final carry-sum pair. In the following we prove this assertion.

Since the function T always underestimates, the result is never over-reduced, i.e.,

$$C_R + S_R \ge 0.$$

If the estimated sign in Step B is positive for all Q iterations, then QN is subtracted from the initial pair; therefore

$$C_R + S_R = C_0 + S_0 - QN < N + 2^t.$$

If the estimated sign becomes negative in an iteration, it stays negative thereafter to the last iteration. Thus, the condition

$$T(\hat{C}) + T(\hat{S}) < 0$$

in the last iteration of Step B implies that

$$T(\hat{C}) + T(\hat{S}) \le -2^t, \tag{4}$$

since $T(X)$ is always a multiple of 2^t. By applying (3) to $\hat{C}$ and $\hat{S}$, we have

$$T(\hat{C}) + T(\hat{S}) \le \hat{C} + \hat{S} < T(\hat{C}) + T(\hat{S}) + 2^{t+1}.$$

It follows from the above equation and (4) that

$$\hat{C} + \hat{S} < 2^{t+1} - 2^t = 2^t.$$

Since in Step A we perform $(\hat{C}, \hat{S}) := C + S - N$ and in the last iteration the carry-sum pair is not reduced (because the estimated sign is negative), we must have

$$C_R + S_R = \hat{C} + \hat{S} + N,$$

which implies

$$C_R + S_R < N + 2^t.$$

The modular reduction procedure described above subtracts N from (C, S) in each of the Q iterations. The procedure can be improved in speed by subtracting $2^{k-j}N$ during iteration j, where $(Q + 1) \le 2^k$ and $j = 1, 2, 3, \ldots, k$. For example, if $Q = 3$, then $k = 2$ can be used. Instead of subtracting N three times, we first subtract $2N$ and then N. This observation is utilized in the following algorithm:

Algorithm B

1. Set $S^{(0)} := 0$ and $C^{(0)} := 0$.
2. Repeat 2a, 2b, and 2c for $i = 1, 2, 3, \ldots, n$
   2a. $(C^{(i)}, S^{(i)} := 2C^{(i-1)} + 2S^{(i-1)} + A_{n-i}B$.
   2b. $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - 2N$.
   If $T(\hat{C}^{(i)} + T(\hat{S}^{(i)}) \ge 0$, then set $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
   2c. $(\hat{C}^{(i)}, \hat{S}^{(i)} := C^{(i)} + S^{(i)}) - N$.
   If $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \ge 0$, then set $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
3. End.

The parameter t controls the precision of estimation; the accuracy of the estimation and the total amount of logic required to implement it decreases as t increases. After Step 2c, we have

$$C^{(i)} + S^{(i)} < N + 2^t,$$

which implies that after the next shift-add step the range of $C^{(i+1)} + S^{(i+1)}$ will be $[0, 3N + 2^{t+1})$. Assuming $Q = 3$, we have

$$3N + 2^{t+1} \le (Q + 1)N + 2^t = 4N + 2^t,$$

which implies $2^t \le N$, or $t \le n - 1$. The range of $C^{(i+1)} + S^{(i+1)}$ becomes

$$0 \le C^{(i+1)} + S^{(i+1)} < 3N + 2^{t+1} \le 3N + 2^n \le 2^{n+2},$$

and after Step 2b, the range will be

$$-2^{n+1} \le -2N \le C^{(i+1)} + S^{(i+1)} < N + 2^n < 2^{n+1}.$$

In order to contain the temporary results, we use $(n + 3)$-bit carry save adders which can represent integers in the range $[-2^{n+2}, 2^{n+2})$. When $t = n - 1$, the sign estimation technique checks 5 most significant bits of $\hat{C}^{(i)}$ and $\hat{S}^{(i)}$ from the bit locations $n - 2$ to $n + 3$.

Algorithm B produces a pair of integers $(C, S) = (C^{(n)}, S^{(n)})$such that $P = C + S$ is in the range $[0, 2N)$. The final result in the correct range $[0, N)$ can be obtained by computing $P = C + S$ and $\hat{P} = C + S - N$ using carry propagate adders. If $\hat{P} < 0$, we have $P = \hat{P} + N < N$, and thus P is in the correct range. Otherwise, we choose $\hat{P}$ because $0 \le \hat{P} = P - N < 2^t < N$ implies $\hat{P} \in [0, N)$.

The steps of Algorithm B computing $47 \cdot 48 \pmod{50}$ are illustrated in figure 1. Here $n = \lfloor \log_2(50) \rfloor + 1 = 6$, and the algorithm computes $(C, S) = (010111000, 110000000) = (184, -128)$ in $3n = 18$ clock cycles. The range of $C + S = 184 - 128 = 56$ is $[0, 2 \cdot 50)$. The final result is found by computing $C + S = 56$ and $C + S - N = 6$, and selecting the latter since it is positive.

3. Semi-Systolic Array

The carry save adder structure shown in figure 2 implements an instance of Step 2. The semi-systolic

		C	S	$\hat{C}$	$\hat{S}$	$T(\hat{C})+T(\hat{S})$	R
$i=0$		000000000	000000000	–	–	–	–
$i=1$	2a	000000000	000110000	–	–	–	–
	2b	000000000	000110000	000100000	110101100	111000000	1
	2c	000000000	000110000	000000000	111111110	111100000	1
$i=2$	2a	000000000	001100000	–	–	–	–
	2b	000000000	001100000	000000000	111111100	111100000	1
	2c	010000000	110101110	010000000	110101110	000100000	0
$i=3$	2a	000100000	001101100	–	–	–	–
	2b	001011000	111010000	001011000	111010000	000000000	0
	2c	001011000	111010000	110110000	001000110	111100000	1
$i=4$	2a	101100000	100100000	–	–	–	–
	2b	001000000	111011100	001000000	111011100	000000000	0
	2c	001000000	111011100	110011000	001010010	111000000	1
$i=5$	2a	101100000	100001000	–	–	–	–
	2b	101100000	100001000	000010000	111110100	111100000	1
	2c	010010000	110100110	010010000	110100110	000100000	0
$i=6$	2a	001000000	001011100	–	–	–	–
	2b	010111000	110000000	010111000	110000000	000100000	0
	2c	010111000	110000000	100010000	011110110	111100000	1

Fig. 1. An example illustrating Algorithm B for $n = 6$. The algorithm computes $P = A \cdot B \pmod N$ where $A = 47 = (000101111)$, $B = 48 = (000110000)$, $N = 50 = (000110010)$, and $M = -N = (111001110)$. The final value of the carry-sum pair is found as $(C, S) = (184, -128)$, which gives $C + S = 184 - 128 = 56 \in [0, 2 \cdot 50)$.

multiplier array is constructed by cascading n copies of this structure. The functions of the cells are also described in figure 2. The bits of B and M $(= -N)$ enter from the north-end in a word-serial, bit-parallel fashion. The skewed bits of A enter the array from the west, however, as soon as A_{n-i} enters, it is broadcast to all $n + 3$ cells in that row. The first row of the structure shown in figure 2 implements Step 2a, while the remaining two rows implement Step 2b and Step 2c, respectively. First, temporary values $\hat{C}^{(i)}$ and $\hat{S}^{(i)}$ are computed. The 3-bit carry look-ahead logic (cell L) receives the most significant 5 bits of $\hat{C}^{(i)}$ and $\hat{S}^{(i)}$ (bit locations $n-2, n-1, \ldots, n+2$ corresponding to Y and U cells) and computes the sign bit R. The sign bit is broadcast to all Y, Z, U, and W cells as soon as it is computed: If the estimated sign is positive (i.e., $R = 0$), then the temporary values are taken to be primary values for the next cycle. At the end of $3n$ clock cycles, the pair (C, S) is produced such that $C + S \in [0, 2N)$.

The array is suitable for computing several modular multiplications given the sequences of integers (A, B, N), (A', B', N'), (A'', B'', N''), and so on. Figure 3 illustrates the use of the semi-systolic array for computing several modular multiplications. At the end of $3n$ clock cycles, the pair (C, S) corresponding to the input sequence A, B, and N exits the array from the south-end. The second result, the pair (C', S') corresponding to the next input sequence A', B', and N', follows the first one after a single clock cycle. The latency of the pipe is equal to $3n$, however, if the pipe is full at all times, the data rate will be equal to the clock rate.

The semi-systolic array requires broadcasting of the bits of A and the sign bit R to all cells in each row. This requirement severely limits high-throughput when n is very large. In the following, we give a design which achieves communication locality at the expense of increasing the latency.

4. Systolic Array

The broadcasting requirement of the above multiplier array can be removed by latching each single bit of A as it enters from the west (and moves horizontally to the east). Additionally, the estimated sign bit R also needs to be latched between the cells. This technique, however, requires a reordering (rescheduling) of the computations performed in the nodes. Techniques have been developed for designing systolic schedules, given some representation of the algorithm (see [13] and the

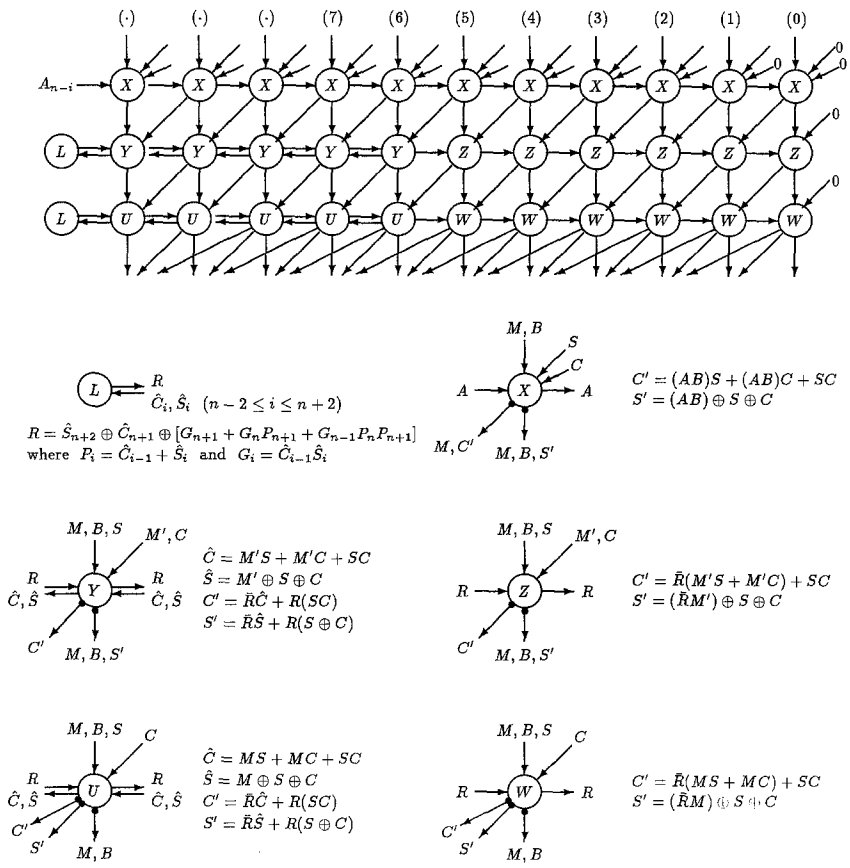

Fig. 2. An instance of the semi-systolic array implementing Blakley's algorithm for $n = 8$. The decimal numbers in parentheses indicate bit locations. The cell definitions also follow, where $\oplus$ stands for exclusive-OR and $\bar{R}$ represents the logical complement of the boolean variable R. The black dots indicate that the variables are latched at the output of a cell. The unlatched variables are the bits of A, the estimated sign bit R, and the most significant 5 bits of $\hat{C}$ and $\hat{S}$.

	$\begin{matrix}(1,0)\\(1,0)\\(1,0)\end{matrix}$	$\begin{matrix}(1,0)\\(1,0)\\(1,0)\end{matrix}$	$\begin{matrix}(1,0)\\(1,0)\\(1,0)\end{matrix}$	$\begin{matrix}(M_3'', B_3'')\\(M_3', B_3')\\(M_3, B_3)\end{matrix}$	$\begin{matrix}(M_2'', B_2'')\\(M_2', B_2')\\(M_2, B_2)\end{matrix}$	$\begin{matrix}(M_1'', B_1'')\\(M_1', B_1')\\(M_1, B_1)\end{matrix}$	$\begin{matrix}(M_0'', B_0'')\\(M_0', B_0')\\(M_0, B_0)\end{matrix}$
A_3'', A_3', A_3	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	1	**1**	**1**	**1**	**1**	**1**	**1**
	2	**2**	**2**	**2**	**2**	**2**	**2**
$A_2'', A_2', A_2 \cdots$	**3**	**3**	**3**	**3**	**3**	**3**	**3**
	4	**4**	**4**	**4**	**4**	**4**	**4**
	5	**5**	**5**	**5**	**5**	**5**	**5**
$A_1'', A_1', A_1 \cdots \cdots$	**6**	**6**	**6**	**6**	**6**	**6**	**6**
	7	**7**	**7**	**7**	**7**	**7**	**7**
	8	**8**	**8**	**8**	**8**	**8**	**8**
$A_0'', A_0', A_0 \cdots \cdots \cdots$	**9**	**9**	**9**	**9**	**9**	**9**	**9**
	10	**10**	**10**	**10**	**10**	**10**	**10**
	11	**11**	**11**	**11**	**11**	**11**	**11**
				$\begin{matrix}(C_3'', S_3'')\\(C_3', S_3')\\(C_3, S_3)\end{matrix}$	$\begin{matrix}(C_2'', S_2'')\\(C_2', S_2')\\(C_2, S_2)\end{matrix}$	$\begin{matrix}(C_1'', S_1'')\\(C_1', S_1')\\(C_1, S_1)\end{matrix}$	$\begin{matrix}(C_0'', S_0'')\\(C_0', S_0')\\(C_0, S_0)\end{matrix}$

Fig. 3. The use of the semi-systolic array of size $n = 4$ to compute several modular multiplications. The array has $3n = 12$ rows of $n + 3 = 7$ simple adder cells, and the latency is equal to $3n = 12$. The boldface numbers show the time steps at which the cells execute, and the dot (·) represents a single delay.

references therein). In general, a systolic schedule (and its corresponding array) is produced by first constructing a localized process dependence dag of the algorithm, and then by mapping the process dependence dag in spacetime.

Since the sign estimation function requires the most significant 5 bits of the intermediate result $(\hat{C}, \hat{S})$, we merge a single L cell (carry look-ahead logic) and 5 left-most Y cells into a single supercell named LY^5. Similarly, a single L and 5 left-most U cells are also merged to obtain a supercell named LU^5. Also, we merge 5 left-most X cells to obtain a single X^5 supercell. The remaining X, Z, and W cells are untouched. This merging process helps to achieve communication locality. After this assignment, the dag contains $2n$ rows where each row has $n - 1$ cells.

We draw the process dependence dag in the (i, j) coordinate system where i is the abscissa (which increases from the east to the west) and j is the ordinate (which increases from the north to the south). In this coordinate system, the top-most row and the right-most column are indexed with $i = 0$ and $j = 0$, respectively. The process dependence dag and the coordinate system are shown in figure 4. The arcs of the dag are given as:

$(-1, 0)$ corresponds to R,
A,

$(0, 1)$ corresponds to B,
S from $j = 0 + 3k, 1 + 3k$,
M from $j = 0 + 3k, 1 + 3k$,
$2 + 3k$,

$(1, 1)$ corresponds to C from $j = 0 + 3k, 1 + 3k$,
S from $j = 2 + 3k$,
M from $j = 0 + 3k$,

$(2, 1)$ corresponds to C from $j = 2 + 3k$,

where $0 \leq k \leq n - 1$. Since arc $(2, 1)$ connects rows $j = 2 + 3k$ and $j = 0 + 3k$ for $k = 0, 1, \ldots, n - 1$, the dag in figure 4 is not *homogeneous* (see [13] for definitions). A homogeneous dag can be obtained by grouping two X cells into a supercell, named X^2. The same process forms Z^2 and W^2 supercells. This grouping transforms arc $(2, 1)$ into $(1, 1)$, thus giving the new set of arcs as

$$E \{(-1, 0), (0, 1), (1, 1)\}.$$

Also note that the width of the dag shrinks to $w = \lceil n/2 \rceil$. The resulting dag is shown in figure 5. The *hyperplanes*, or the equi-temporal hyperplanes, are parallel planes on the process dependence dag such that all processes (nodes) on the same hyperplane are processed at the same time [13]. The *schedule vector* is the

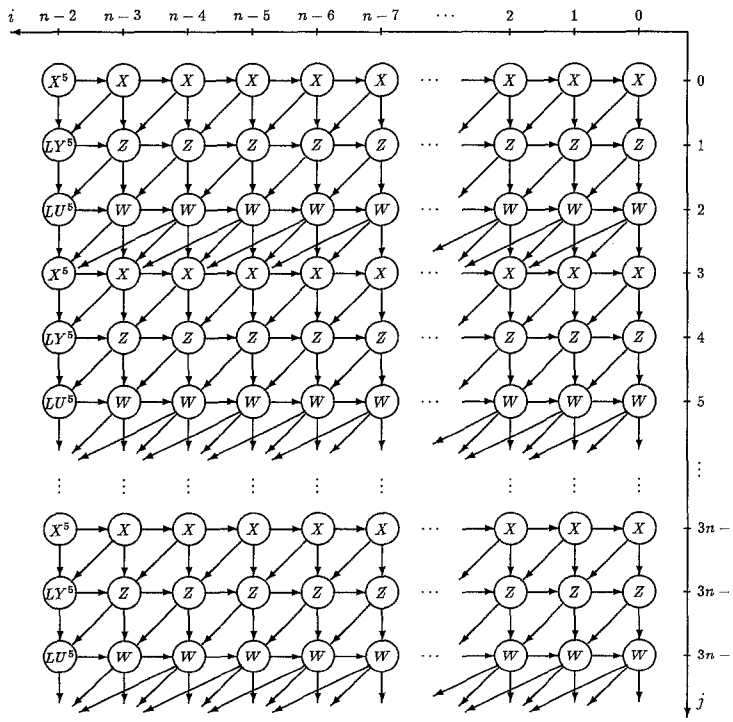

Fig. 4. The process dependence dag drawn in the coordinate system (i, j). The dag is not homogeneous due to the arcs between some cells, e.g., (0, 2) and (2, 3).

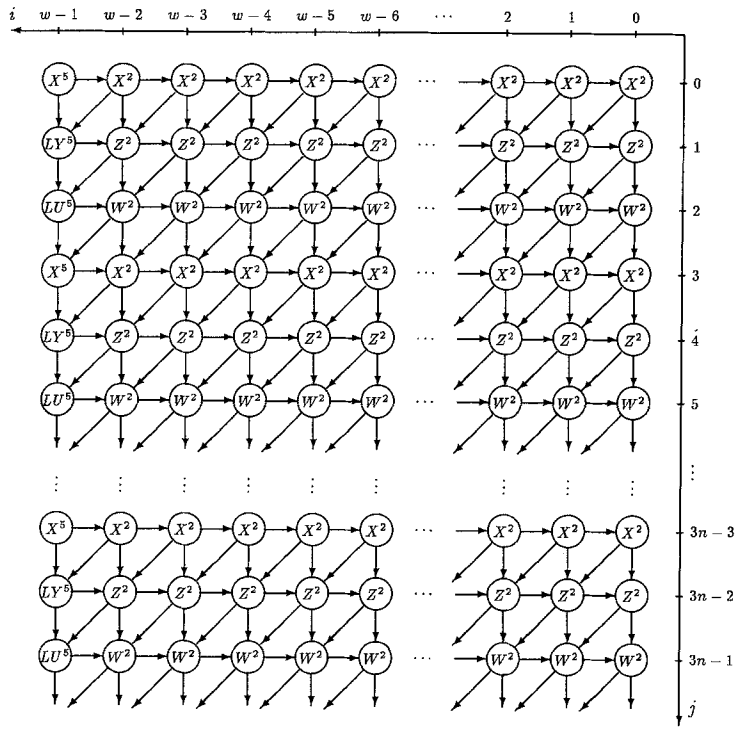

Fig. 5. The homogeneous process dependence dag drawn in the coordinate system (i, j).

vector normal to the hyperplanes, which must obey causality and local connectivity, i.e., for all $(i, j) \in E$,

$$ai + bj > 0,$$

where a and b are the components of the schedule vector (a, b). The above inequality implies

$$\begin{aligned} a \cdot (-1) + b \cdot (0) &= -a > 0 \\ a \cdot (1) + b \cdot (0) &= b > 0 \\ a \cdot (1) + b \cdot (1) &= a + b > 0. \end{aligned}$$

Thus, the components of a permissible schedule vector, i.e., the integers a and b, must satisfy

$$a < 0 \text{ and } b > |a|. \tag{5}$$

The optimal values of (a, b), are those for which the *latency* of the systolic array is minimum. The schedule function $t(i, j)$ gives the time step (clock cycle) t at which the process (i, j) is executed. It is written as

$$t(i, j) = ai + bj + c,$$

where c is an integer constant. The latency of the systolic array is the total number of cycles required to compute the entire array of processes:

$$\begin{aligned} L &= t_{last} - t_{first} + 1 = t(0, 3n - 1) - t(w - 1, 0) \\ &\quad + 1 = b \cdot (3n - 1) - a \cdot (w - 1) + 1. \end{aligned}$$

The optimal values of a and b satisfy the constraints given by (5) and minimize the above equation. It is easily found by enumeration that $(a, b) = (-1, 2)$, which gives the minimum value of the latency as

$$L_{min} = 6n + w - 2. \tag{6}$$

The constant value c is determined by scheduling the first node to be processed to time step zero. The node indexed with $(i, j) = (w - 1, 0)$ is the first process to be executed, i.e.,

$$t(w - 1, 0) = -(w - 1) + c = 0,$$

which gives $c = w - 1$, thus the schedule function is found as

$$t(i, j) = -i + 2j + w - 1. \tag{7}$$

We should add that the above schedule is by no means the only one: there exist many (linear or nonlinear) schedules corresponding to the process dependence dag given in figure 5. However, the linear schedule given by (7) is simple and easy to implement. Figure 6 illustrates the input and output pattern of the systolic array computing several modular multiplications.

	$\cdot$	$\cdot$	$(M_1'', B_1'', M_0'', B_0'')$
	$\cdot$	$(M_3'', B_3'', M_2'', B_2'')$	(M_1', B_1', M_0', B_0')
	$(M_5'', B_5'', M_4'', B_4'')$	(M_3', B_3', M_2', B_2')	(M_1, B_1, M_0, B_0)
	(M_5', B_5', M_4', B_4')	(M_3, B_3, M_2, B_2)	$\cdot$
	(M_5, B_5, M_4, B_4)	$\cdot$	$\cdot$
A_5'', A_5', A_5	0	1	2
	2	3	4
	4	5	6
A_4'', A_4', A_4 ·····	6	7	8
	8	9	10
	10	11	12
A_3'', A_3', A_3 ····· ·····	12	13	14
	14	15	16
	16	17	18
A_2'', A_2', A_2 ····· ····· ·····	18	19	20
	20	21	22
	22	23	24
A_1'', A_1', A_1 ····· ····· ····· ·····	24	25	26
	26	27	28
	28	29	30
A_0'', A_0', A_0 ····· ····· ····· ····· ·····	30	31	32
	32	33	34
	34	35	36
	$\cdot$	$\cdot$	$(C_1'', S_1'', C_0'', S_0'')$
	$\cdot$	$(C_3'', S_3'', C_2'', S_2'')$	(C_1', S_1', C_0', S_0')
	$(C_5'', S_5'', C_4'', S_4'')$	(C_3', S_3', C_2', S_2')	(C_1, S_1, C_0, S_0)
	(C_5', S_5', C_4', S_4')	(C_3, S_3, C_2, S_2)	$\cdot$
	(C_5, S_5, C_4, S_4)	$\cdot$	$\cdot$

Fig. 6. The use of the systolic array of size $n = 6$ to compute several modular multiplications. The array has $3n = 18$ rows of $w = \lceil n/2 \rceil = 3$ adder cells, and the latency is equal to $6n + w - 2 = 37$. The boldface numbers show the time steps at which the cells execute, and the dot ($\cdot$) represents a single delay.

5. Conclusions

We have presented a bit-level systolic array implementing Blakley's algorithm for modular multiplication. It consists of $3nw$ small adder cells and computes a single modular multiplication in $6n + w - 2$ clock cycles. Compared to the semi-systolic array, the systolic version requires more delay elements and has longer latency in terms of clock cycles: $6n + \lceil n/2 \rceil - 2$ versus $3n$. However, the systolic array is free of broadcast, and therefore, can operate with a much higher clock rate.

This array can be used to implement certain interactive computation protocols and cryptosystems. The RSA algorithm requires the computation of modular exponentiation which is broken into a succession of modular multiplications by the application of the binary method [14]. A survey of hardware implementations of the RSA algorithm is given in [15]. A systolic array implementing the RSA algorithm, which uses systolic modular multipliers as building blocks, was reported in [16]. The systolic modular multiplier presented in this paper is suitable for such implementation.

As a future topic of study, we point out an algorithm by Brickell for implementing modular multiplication using delayed carry save adders [17]. Instead of (1), Brickell's technique computes $2^{10}P = 2^{10}AB \pmod{2^{10}N}$ in $n + 7$ clock cycles using $n + 11$ arithmetic cells. Recently, it was noted by Forster et al. in [18] that Koç and Hung's sign estimation algorithm and Brickell's delayed carry save adder technique can be utilized to perform modular multiplication in only n time steps. Thus, it seems worthwhile to investigate whether this strategy would yield efficient systolic arrays implementing Brickell's algorithm.

References

1. R.L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM*, 21, 1978, pp. 120–126.
2. A.G. Konheim, *Cryptography, A Primer.* New York: John Wiley, 1981.
3. N. Koblitz, *"A Course in Number Theory and Cryptography,* New York: Springer-Verlag, 1987.

4. M.A. Soderstrand, W.K. Jenkins, G.A. Jullien, and F.J. Taylor, (ed.), *Residue Arithmetic: Modern Applications in Digital Signal Processing*. New York: IEEE Press, 1986.
5. K. Hwang, *Computer Arithmetic, Principles, Architecture, and Design*. New York: John Wiley, 1979.
6. E.E. Swartzlander, (ed.), *Computer Arithmetic*, vol. I and II. Los Alamitos: IEEE Computer Society Press, 1990.
7. D. Simmons and S.E. Tavares, "An NMOS implementation of a large number modulo multiplier for data encryption systems." In *Proceedings of the 1983 IEEE Custom Integrated Circuits Conference*, Rochester, New York: IEEE Press, 1983, pp. 262–266.
8. G.R. Blakley, "A computer algorithm for the product AB modulo M," *IEEE Transactions on Computers*, 32, 1983, pp. 497–500.
9. K.R. Sloan, Jr. "Comments on: 'A computer algorithm for the product AB modulo M.' " *IEEE Transactions on Computers*, 34, 1985, pp. 290–292.
10. P.W. Baker, "Fast computation of $A * B$ modulo N," *Electronics Letters*, 23, 1987, pp. 794–795.
11. Ç.K. Koç and C.Y. Hung, "Multi-operand modulo addition using carry save adders." *Electronics Letters*, 26, 1990, pp. 361–363.
12. Ç.K. Koç and C.Y. Hung, "Carry save adders for computing the product AB modulo N." *Electronics Letters*, 26, 1990, pp. 899–900.
13. S.Y. Kung, "*VLSI Array Processors*." Englewood Cliffs, NJ: Prentice-Hall, 1988.
14. D.E. Knuth. "*The Art of Computer Programming: Seminumerical Algorithms*," vol. 2. Reading, MA: Addison-Wesley, (2nd ed.), 1981.
15. E.F. Brickell, "A survey of hardware implementations of RSA." In (G. Brassard, ed.) *Advances in Cryptology—Crypto 89, Proceedings*, Lecture Notes in Computer Science, No. 435, New York: Springer-Verlag, 1989, pp. 368–370.
16. C.N. Zhang, H.L. Martin, and D.Y.Y. Yun, "Parallel algorithms and systolic arrays designs for RSA cryptosystem." In (K. Bromley, S.Y. Kung, and E. Swartzlander, ed., *Proceedings of the International Conference on Systolic Arrays*, San Diego, California, Los Alamitos: IEEE Computer Society Press, 1988, pp. 341–350.
17. E.F. Brickell, "A fast modular multiplication algorithm with application to two key cryptography." In (D. Chaum, R.L. Rivest, and A.T. Sherman, ed.) *Advances in Cryptology, Proceedings of Crypto 82*, New York: Plenum Press, 1982, pp. 51–60.
18. C.H.N. Forster, S.S. Dlay, and R.N. Gorgui-Naguib, "Carry delayed save adders for computing the product AB modulo N in $\log_2 N$ steps." *Electronics Letters*, 26, 1990, pp. 1544–1545.

Çetin Kaya Koç has been an Assistant Professor in the Department of Electrical Engineering at the University of Houston since June 1988. He received his B.S. (1980, Highest Honors) and M.S. (1982) degrees in Electrical Engineering from Istanbul Technical University and his M.S. (1985) and Ph.D. (1988) degrees in Electrical and Computer Engineering from the University of California, Santa Barbara. His research interests include parallel computation, computer arithmetic, cryptography, and scientific computing.

Ching Yu Hung received his B.S. degree from the National Taiwan University in 1986 and his M.S. degree from the University of Houston in December 1990, both in Electrical Engineering. He is now a graduate student pursuing a Ph.D. degree in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. His research interests include computer arithmetic, parallel architectures, and algorithms.