Two systolic architectures for multiplication in $GF(2^m)$

W.C. Tsai and S.-J. Wang

Abstract: Two new systolic architectures are presented for multiplications in the finite field $GF(2^m)$. These two architectures are based on the standard basis representation. In Architecture-I, the authors attempt to speed up the operation by using a new partitioning scheme for the basic cell in a straightforward systolic architecture to shorten the clock cycle period. In Architecture-II, they eliminate the one clock cycle gap between iterations by pairing off the cells of Architecture-I. They compare their architectures with previously proposed systolic architectures and a semi-systolic architecture, and show that their Architecture-I offers the highest speed and Architecture-II the lowest hardware complexity.

1 Introduction

In recent years, the finite field has been widely used in various data communication applications such as switching theory, error correction coding, pseudo-random number generation and cryptosystems, [1, 2]. A high-speed and low-complexity design for finite field arithmetic is very necessary for meeting the demands of wider bandwidths, better security, and higher portability for personal communication.

In these applications, the Galois field of order $q = p^m$, denoted as $GF(p^m)$, is usually used, especially in the case $p = 2$. Addition over $GF(2^m)$ can be easily implemented by bit-wise exclusive-OR without any carry propagation problem. Multiplication over $GF(2^m)$, on the other hand, is much more complex. In the implementation of multiplication over $GF(2^m)$, the design may use standard basis, normal basis, or dual basis representation. In this paper, we will propose two high-speed and low-complexity systolic architectures based on standard basis representation (SBR).

For a high-speed multiplier over $GF(2^m)$, several designs [3–5] adopting the architecture of semi-systolic arrays have been proposed. However, all these semi-systolic architectures have to broadcast some global signals. It becomes more difficult to handle the broadcasting problem as the bit length $m$ becomes larger. On the other hand, due to the regularity of cells and the locality of connections, a ‘pure’ systolic array, instead of a semi-systolic array, is usually a more appropriate choice for VLSI implementation [6–13]. These systolic architectures usually decompose multiplication over $GF(2^m)$ into a sequence of additions to sum up partial products, and modular operations to perform SBR conversion. We may classify these systolic designs into two categories according to their computation procedure: (1) summing first and (2) modulus first. The ‘summing first’ designs perform the addition first and then covert the sum into SBR form. Within this category, Wang et al. [7] implemented $AB$ in a straightforward way, Wei [8] implemented $AB^2 + C$, Guo et al. [9] used a high-radix implementation for computing $AB$, and Mekhallalati et al. [10] slightly modified the systolic architecture in a semi-systolic array by applying a re-timing process to reduce the initial delay. Conversely, the ‘modulus first’ designs convert both the addend and the augend into SBR form first and then perform the addition. Within this category, Yeh et al. [6] implemented an architecture to compute $AB + C$, Ghafoor et al. [11] adopted the same architecture to perform the exponentiation operation, and Hasan et al. [12] arranged the converted addend and augend in a matrix form so that all the computations of multiplication, division, and inversion can be treated as matrix operations.

Among all these architectures, Yeh’s design [6] is the fastest due to its shortest clock period, while one of Mekhallalati’s designs (Systolic-II) [10] has a superior performance in the area-time product. In Yeh’s architecture, the main operation is decomposed into two parallel operations to shorten the clock period and some flip-flops are inserted in the architecture to avoid the inherent one-clock-cycle-gap problem of a bit-by-bit systolic array. However, the partitioning scheme needs one extra control signal, and the insertion of flip-flops increases both the area and power consumption. In Mekhallalati’s design, re-timing is applied on the connections between the $i$th cell and the $(i + 1)$th cell of a semi-systolic architecture, where $i$ is odd, to avoid the one-clock-cycle-gap problem and to reduce the latency down to $m$ clock cycles, where $m$ is the degree of the finite field $GF(2^m)$. In this design, a circuit-level optimisation is also applied on the cells to shorten the clock period.

In this paper, we propose two new architectures, Architecture-I and Architecture-II, to further improve the operation speed and to reduce the area complexity. Architecture-I effects the partitioning on the general cells in Kung’s design [14] to shorten the clock period. Architecture-II is constructed by pairing off the cells in Architecture-I to reduce the latency. As will be shown in Section 4, the partitioning of cells makes Architecture-I one of the fastest
designs for computing independent multiplications, while the alliance of partitioning and pairing makes Architecture-II the fastest design for computing dependent multiplications. Moreover, Architecture-II has the lowest area-time complexity no matter whether the computed multiplications are dependent or independent.

2 Systolic Architecture-I

$GF(2^m)$, an extension field of $GF(2)$, contains $2^m$ elements and a special polynomial $F(x)$. Here, $F(x)$ is a monic, irreducible polynomial over $GF(2)$ of degree $m$ and can be expressed as

$$F(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$$

$$= x^m + f_{m-1} x^{m-1} + f_{m-2} x^{m-2} + \ldots + f_0 x + f_0,$$  \hspace{1cm} (1)

where $f_i$ is either 0 or 1.

If $x$ is a root of $F(x)$, the set $\{1, x, x^2, \ldots, x^{m-1}\}$ forms the standard basis of $GF(2^m)$. For any two elements $A(x)$ and $B(x) \in GF(2^m)$, they can be expressed in SBR form as:

$$A(x) = \sum_{i=0}^{m-1} a_i x^i = a_{m-1} x^{m-1} + a_{m-2} x^{m-2} + \ldots + a_1 x + a_0,$$  \hspace{1cm} (2)

and

$$B(x) = \sum_{i=0}^{m-1} b_i x^i = b_{m-1} x^{m-1} + b_{m-2} x^{m-2} + \ldots + b_1 x + b_0,$$  \hspace{1cm} (3)

where $a_i$ and $b_i$ are binary numbers.

The multiplication of $A$ and $B$ can be computed by multiplying $A(x)$ with $B(x)$ first and then performing (modulo $F(x)$) to convert the product back to the SBR form. An algorithm for computing the multiplication $P(x) = A(x) \times B(x)$ over $GF(2^m)$ can be expressed as:

**Multiplication algorithm over $GF(2^m)$ by using modulus operation**

\begin{align*}
R^0(x) &= 0; \\
\text{for } i = 1 \text{ to } m &\quad R^i(x) = (R^{i-1}(x) x + a_{m-i} B(x)) \pmod{F(x)}; \\
\text{end.} &\quad P(x) = R^m(x),
\end{align*}

where $a_j$ is the $j$th coefficient of $A(x)$, $R^j(x) = \sum_{j=0}^{m-1} r_j x^j$ is the partial sum after the $j$th iteration, and $a_{m-j} B(x) = \sum_{j=0}^{m-1} (a_{m-j} b_j) x^j$.

The main operation, $R^i(x) = (R^{i-1}(x) x + a_{m-i} B(x)) \pmod{F(x)}$, of the above algorithm can be rewritten as $R^i(x) = (R^{i-1}(x) x \pmod{F(x)}) + a_{m-i} B(x)$. This is because $a_{m-i} B(x)$ is already in SBR form. Hence, the computation of $R^i(x)$ can be treated as the combination of a modular operation and an addition. The modular operation $(R^{i-1}(x) x \pmod{F(x)})$, can be computed by converting the highest order term of $(R^{i-1}(x) x)$ into the SBR form first, and then adding the converted result with the remaining part of $(R^{i-1}(x))$. That is,

$$(R^{i-1}(x) x \pmod{F(x)}) = \sum_{k=0}^{m-1} r_k x^{k+1},$$

$$= r'_{m-1} x^m + \sum_{k=0}^{m-2} r_k x^{k+1},$$

$$= r'_{m-1} x^m + \sum_{j=0}^{m-1} f_j x^j + \sum_{j=1}^{m-1} r_j^{-1} x^j,$$

$$= \sum_{j=0}^{m-1} (r'_{m-1} f_j + r_j^{-1} x^j) \pmod{F(x)}.$$

At the bit level, the algorithm becomes

**Bit-level multiplication algorithm over $GF(2^m)$**

$R^0(x) = 0$; $r_k^{-1} = 1$, $k = 1$ to $m$;

for $i = 1$ to $m$

$$R^i(x) = \sum_{j=0}^{m-1} (r'_{m-1} f_j \oplus r_j^{-1} \oplus a_{m-j} b_j) x^j;$$

end

$$P(x) = R^m(x).$$

In the above algorithm, the main operation can be computed bit-by-bit by adding three operands: $r'_{m-1} f_j$, $r_j^{-1}$, and $a_{m-j} b_j$. Because $j^{-1}$, the MSB of $R^{j-1}(x)$, is involved in the computation of $r_j$ for all $j$, it is more efficient to compute $R(x)$ with the most significant bit calculated first. Hence, in the above algorithm, we compute $R(x)$ starting from the MSB toward the LSB. A 2D systolic architecture for the implementation of this multiplication algorithm is illustrated in Fig. 1. In this figure, the data dependency between bits and between iterations is shown. The cell on the $j$th column and $i$th row computes the $j$th bit of $R(x)$ by computing

$$r_j = r'_{m-1} f_j \oplus r_j^{-1} \oplus a_{m-j} b_j,$$  \hspace{1cm} (4)

where $r_j^{-1}$ is the most significant coefficient of $R^{j-1}(x)$. By applying vertical projection on this 2D systolic array, we get the 1D systolic array as shown in Fig. 2. The timing sequence of $r_j$‘s is illustrated in Fig. 3. Note that the delay time between successive iterations is two clock cycles; there is a one clock cycle gap between $r_{m-1}^{-1}$ and $r_{m-1}$, a clock cycle gap between $r_{m-1}$ and $r_{m-1}$, and so on. This is due to the inherent characteristics of the (mod $F(x)$) operation in the multiplication algorithm, as will be stated in the next paragraph.

In eqn. 4, it can be seen that $r_j^{-1}$ is required to compute $r_j$. Consider we want to compute $R(x)$ bit-by-bit. After the computation of $r_j^{-1}$, we need one more clock cycle to calculate $r_j^{-1}$ before the computation of $r_j$. That is to say, there exists one clock cycle delay between the computation of the same order coefficient in two adjacent iterations. Hence, the average computation time of this architecture for $N$ m-bit multiplications over $GF(2^m)$ becomes $2N$ clock cycles: $m N$ cycles to operate plus $m N$ interlaced clock cycles to wait. To further improve the performance of the architecture, these idling clock cycles could be utilised to compute another independent operation without any time conflict; i.e. this bit-by-bit architecture can achieve the performance of $m$ clock cycles per operation when computing independent multiplications.
In this paper, we propose a new architecture to further improve the computation speed by partitioning the main operation of the bit-level algorithm. In eqn. 4, note that \( f_j, a_{m-i}, \) and \( b_j \) could be available in advance. As \( r_{m-1}^i \) is ready, \( r_{m-1}^{i-1} f_j \oplus a_{m-i} b_j \) can be calculated immediately.

\[
\begin{align*}
\text{Hence, the computation of } r_j^i \text{ can be partitioned into two simpler operations to shorten the clock period. These two operations are:} \\
p_j^i &= r_{m-1}^{i-1} f_j \oplus a_{m-i} b_j, \\
r_j^i &= r_{j-1}^{i-1} \oplus p_j^i.
\end{align*}
\]

This partitioned architecture, called Architecture-I, is shown in Fig. 4 and its 1D systolic architecture is shown in Fig. 5. Two kinds of simple cells are used to calculate \( p_j^i \) and \( r_j^i \), respectively; the upper layer cells compute \( p_j^i \) while the lower layer cells compute \( r_j^i \).

The operation of Architecture-I can be expressed in the following algorithm:

\[
\begin{align*}
&\text{for } k = 0 \text{ to } m - 1, \\
&\quad \text{for } i = 1 \text{ to } m \\
&\quad \quad \text{in parallel} \\
&\quad \quad r_k^0 = 0, \\
&\quad \quad r_{k-1}^1 = 0, \\
&\quad \quad \text{for } i = 1 \text{ to } m \\
&\quad \quad \quad \text{in parallel} \\
&\quad \quad \quad p_j^i = r_{m-1}^{i-1} f_j \oplus a_{m-i} b_j, \\
&\quad \quad \quad r_j^i = r_{j-1}^{i-1} \oplus p_j^i, \\
&\quad \quad \quad \text{end in parallel} \\
&\quad \quad \quad \text{end in parallel} \\
&\quad \quad \quad \text{end.}
\end{align*}
\]
The timing sequence of $p_k$'s and $r_k$'s is shown in Fig. 6. The delay time between successive iterations is still two clock cycles, while the clock period has now been shortened due to the partitioning operation. Note that Architecture-I can also achieve full utilisation when calculating independent multiplications over $GF(2^m)$. The clock period before partitioning is the delay of two XOR gates and one AND gate. After partitioning, the clock period becomes the delay of one XOR gate and one AND gate. Since the computation of $p_i$ and $r_j$ can be done in parallel, the average number of clock cycles per multiplication is not changed. The detailed comparison of Architecture-I and some other architectures will be presented later, in Section 4.

3 Systolic Architecture-II

As mentioned before, even though Architecture-I can compute a sequence of independent multiplications with full utilisation, it can only achieve 50% utilisation for a sequence of dependent multiplications. This is because the dependence between multiplications precludes the possibility of interlaced computations. In this section, we propose another architecture, Architecture-II, which is more efficient than Architecture-I when computing dependent multiplications. In Architecture-II, we use cell merging in order to calculate some of the operations before the hand. These pre-computed operations make the removal of the idling cycles in Architecture-I possible. This removal of idling cycles can thus increase the computation efficiency when dealing with dependent multiplications.

As mentioned before, there is a one-clock-cycle-gap problem in a bit-by-bit architecture. Our Architecture-I is basically a bit-by-bit architecture, and the one-clock-cycle-gap problem does exist in that architecture. To avoid this problem, we merge the cells in Architecture-I in a specific way, as shown in Fig. 7. In this figure, we group $r_i$, $r_{i+1}$, $r_{i+2}$, and $p_{i+1}$ together, $r_{i+1}$, $r_{i+2}$, $r_{i+3}$, $p_{i+2}$, and $p_{i+3}$ together, and so on. With this arrangement, if $r_i$, $r_{i+1}$, $p_{i+1}$, and $p_{i+2}$ are computed in the $k$th clock cycle and $r_{i+2}$, $r_{i+3}$, $p_{i+3}$, and $p_{i+4}$ are computed in the $(k+1)$th clock cycle, then $r_{i+1}$, $p_{i+3}$, and $p_{i+4}$ can also be calculated in the $(k+1)$th clock cycle. It looks infeasible, at first glance, to have $r_{i+1}$ calculated in the $(k+1)$th clock cycle, since the computation of $r_{i+1}$ depends on $r_{i+2}$. However, assume we remove the latch between $r_{i+2}$ and $r_{i+3}$, i.e., in the $(k+1)$th clock cycle, $r_{i+1}$ is to be computed immediately after $r_{i+2}$ is computed. Then, all $r_{i+1}$, $r_{i+2}$, $p_{i+3}$, and $p_{i+4}$ can be calculated in the $(k+1)$th clock cycle without a time conflict. Similarly, we remove the latch between $r_{i+4}$ and $r_{i+5}$, the latch between $r_{i+6}$ and $r_{i+7}$, and so on. The removal of these latches only forms a local data propagation and won’t create any global propagation path. Moreover, after merging the cells, we can remove half of the latches that have been used to keep $a_i$'s, $r_{m-1}$'s, and the control signals. This removal can save a huge amount of latches (Fig. 8). The area and power consumption of the architecture can thus be greatly reduced after eliminating these latches.

After merging the cells, the $k$th general cell of Architecture-II in iteration $i$ computes the following four operations:

\[ r'_{j} = r_{j-1} \oplus p'_{j}, \]  
\[ r'_{j-1} = r_{j-2} \oplus p'_{j-1}, \]  
\[ p_{j+1} = r'_{m-1} \oplus a_{m-1} \oplus b_j, \]
and
\[ p_{j+1}^{i} = r_{m-1}^{i} f_{j-1} + a_{m-i-1} b_{j-1}, \]  
where \( j = 2k \).

The operation of Architecture-II can be expressed in the following algorithm (for \( m \) is odd):

\begin{align*}
& r_0^{-1} = 0, \quad k = 0 \text{ to } m - 1; \\
& p_0^0 = 0, \quad k = 0 \text{ to } m - 1; \\
& r_{k}^{-1} = 0, \quad k = 0 \text{ to } m; \\
& \text{for } i = 0 \text{ to } m \\
& \quad \text{in parallel} \\
& \quad \text{for } k = \lceil (m - 1)/2 \rceil \text{ to } 0 \\
& \quad \quad j = 2k + 1, \\
& \quad \quad \text{in parallel} \\
& \quad \quad r_{j}^{-1} = r_{j-1}^{i} \oplus p_{j}^{i}, \\
& \quad \quad r_{j-1}^{-1} = r_{j-2}^{i} \oplus p_{j-1}^{i-1} \\
& \quad \quad p_{j+1}^{i} = r_{m-1}^{j} + a_{m-i-1} b_{j}, \\
& \quad \quad p_{j+1}^{i} = r_{m-1}^{j} f_{j-1} + a_{m-i-1} b_{j-1}, \\
& \text{end in parallel} \\
& \text{end}.
\end{align*}

Figs. 7 and 8 also show the implementation of this algorithm. The upper layer cells compute \( r_{j}^{-1} \) and \( r_{j-1}^{-1} \), while the lower layer cells compute \( p_{j}^{i} \) and \( p_{j-1}^{i-1} \). The computation sequence is shown in Fig. 9. In the MSB cell, \( r_{m-1}^{i} \) is calculated and then immediately used for the
Fig. 9 Timing sequence of $p_i$'s and $r_i$'s

computation of $p_{m-1}^{i+1}$ and $p_{m-2}^{i+1}$, according to eqns. 9 and 10. This temporal dependency, which includes one XOR operation for $r_{m-1}^i$ and one AND plus one XOR operation for $p_{m-1}^{i+1}$ or $p_{m-2}^{i+1}$, forms the critical path of Architecture-II. Since there is no one-clock-cycle-gap problem in Architecture-II, we can compute $N$ dependent $m$-bit multiplications in $mN$ clock cycles with the clock period being about one AND and two XOR gate delay. Conversely, Architecture-I computes $N$ dependent $m$-bit multiplications in $2mN$ clock cycles with a clock period of about one AND and one XOR gate delay. Therefore, for computing dependent multiplications, Architecture-II is superior to Architecture-I in computation speed, area size, and power consumption. Detailed comparisons of Architecture-II to some other architectures will be presented in the next section.

4 Comparisons for multiplication over $GF(2^m)$

In this section, we compare our designs with several systolic arrays for their performance in computing multiplications over $GF(2^m)$. For a fair comparison, we add serial output circuits to our architectures, as shown in Fig. 10a and b, and add a control signal to reset all flip-flops.

![Fig. 10](image-url) The two new architectures with serial output circuits

a Architecture-I

b Architecture-II

Table 1: Comparisons for computing multiplications

<table>
<thead>
<tr>
<th>Author</th>
<th>Cycles</th>
<th>Delay per cell, ns</th>
<th>Total ns</th>
<th>AT (m² ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>area per cell, gate count</td>
<td>total area</td>
<td></td>
</tr>
<tr>
<td>Yeh et al. [6]</td>
<td>$m^*$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{latch}} \approx 1.66$</td>
<td>$\approx 1.66m^*$</td>
<td>150*</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>$3A_{\text{and}} + 2A_{\text{xor}} + 10A_{\text{latch}} + A_{\text{mux}} \approx 90.5$</td>
<td>90.5m</td>
<td></td>
</tr>
<tr>
<td>Wang et al. [7]</td>
<td>$m^*$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{latch}} \approx 1.84$</td>
<td>$\approx 1.84m^*$</td>
<td>163*</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>$3A_{\text{and}} + A_{\text{xor}} + A_{\text{mux}} + 10A_{\text{latch}} \approx 88.5$</td>
<td>88.5m</td>
<td></td>
</tr>
<tr>
<td>Hasan et al. [12]</td>
<td>$m^*$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{latch}} \approx 1.66$</td>
<td>$\approx 1.66m^*$</td>
<td>163*</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>$3A_{\text{and}} + 2A_{\text{xor}} + 11A_{\text{latch}} \approx 98$</td>
<td>98m</td>
<td></td>
</tr>
<tr>
<td>Systolic-Ib [10]</td>
<td>$2m$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{mux}} + T_{\text{latch}} \approx 2.65$</td>
<td>$\approx 5.3m$</td>
<td>182</td>
</tr>
<tr>
<td></td>
<td>0.5$m$</td>
<td>$4A_{\text{and}} + A_{\text{xor}} + 7A_{\text{latch}} \approx 68.5$</td>
<td>34.3m</td>
<td></td>
</tr>
<tr>
<td>Systolic-II [10]</td>
<td>$m$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{mux}} + T_{\text{latch}} \approx 3.1$</td>
<td>$\approx 3.1m$</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>0.5$m$</td>
<td>$5A_{\text{and}} + A_{\text{xor}} + A_{\text{mux}} + 9A_{\text{mux}} + 9A_{\text{latch}} \approx 91$</td>
<td>45.5m</td>
<td></td>
</tr>
<tr>
<td>Architecture-I</td>
<td>$m^*$</td>
<td>$T_{\text{and}} + T_{\text{xor}} + T_{\text{latch}} \approx 1.66$</td>
<td>$\approx 1.66m^*$</td>
<td>123*</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>$2A_{\text{and}} + 2A_{\text{xor}} + A_{\text{mux}} + 8A_{\text{latch}} \approx 74$</td>
<td>74m</td>
<td></td>
</tr>
<tr>
<td>Architecture-II</td>
<td>$m$</td>
<td>$T_{\text{and}} + 2T_{\text{xor}} + T_{\text{latch}} \approx 2.12$</td>
<td>$\approx 2.12m$</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>0.5$m$</td>
<td>$4A_{\text{and}} + 4A_{\text{xor}} + 2A_{\text{mux}} + 9A_{\text{latch}} \approx 95.5$</td>
<td>47.8m</td>
<td></td>
</tr>
</tbody>
</table>

$T_{\text{and}}$: delay of a two-input AND gate (0.33 ns).
$T_{\text{xor}}$: delay of a two-input XOR gate (0.46 ns).
$T_{\text{mux}}$: delay of a two-input Multiplex gate (0.45 ns).
$T_{\text{dmux}}$: delay of a two-output Demultiplex gate (0.45 ns).
$T_{\text{fXOR}}$: delay of a three-input XOR gate (0.64 ns).
$T_{\text{fMux}}$: delay of a five-input XOR gate (1 ns).
$T_{\text{latch}}$: delay of a latch gate (0.87 ns).

* : double if the dependent multiplications are computed.

$A_{\text{and}}$: area of a two-input AND gate (1.5).
$A_{\text{xor}}$: area of a two-input XOR gate (4).
$A_{\text{mux}}$: area of a two-input Multiplex gate (3).
$A_{\text{dmux}}$: area of a two-output Demultiplex gate (3).
$A_{\text{fXOR}}$: area of a three-input XOR gate (6).
$A_{\text{fMux}}$: area of a five-input XOR gate (10).
$A_{\text{latch}}$: area of a latch gate (7.5).

$m$ is the order of the finite field $GF(2^m)$. 
The comparisons of the average speed for computing a sequence of dependent and independent multiplications are shown in Table 1. In this table, all the bit-by-bit structures (including [6, 7, 12] and Architecture-I) need $2m$ clock cycles per operation for dependent multiplications, and $m$ clock cycles per operation for independent multiplications. On the other hand, for most architectures which are not bit-by-bit structures, like our Architecture-II, and Mekhallalati's systolic-II design [10], the average time becomes $m$ clock cycles per operation for both dependent and independent multiplications. Note that, however, Mekhallalati's systolic-Ib design [10] needs $2m$ clock cycles per operation for both dependent and independent multiplications. This is due to the fact that when this architecture is computing a multiplication, no other multiplication can be computed in parallel.

Among the bit-by-bit structures Architecture-I uses partitioning to shorten the clock period. In Yeh's design [6] and Hasan’s design [12], different partition methods are adopted. On the other hand, for those structures which are not bit-by-bit architecture, the clock period is lengthened after merging. In both of Mekhallalati’s designs [10], the cells are merged via the re-timing process of a semi-systolic array, and the expanded clock period is shortened by applying optimisation to the merged cells. In our Architecture-II, however, the cells are merged from Archi-

---

**Fig. 11** Hardware implementation of $M^E$ (mod $F(a)$) with the Architecture-I structure

**Fig. 12** Hardware implementation of $M^E$ (mod $F(a)$) with the Architecture-II structure
tecture-I, which has applied partitioning on the general cells in Fig. 1. Due to the finer structure in Architecture-I, it is easier, and there is more flexibility to carry out the merging while keeping a balanced pipeline structure. The timing and area estimation is based on the delay and gate count information of a TSMC 0.35μm cell library. Since there is no 5-input XOR gate in the library, we estimate its delay and gate count ourselves. In Table 1 we can see that the delay in [6, 15] is similar to the delay of Architecture-I. This is because all these designs have applied partitioning on their architectures. However, due to their complicated methods of handling the one-clock-cycle-gap problem, Yeh’s and Hasan’s designs [6, 15] consume a larger area than our Architecture-I. Table 1 shows that the computation speed of our Architecture-II is the fastest when calculating independent multiplications. Moreover, as mentioned before, a large number of latches can be removed after merging. Therefore, the area size and power consumption are greatly reduced in Mekhallalati’s designs and our Architecture-II design. In Table 1, we can also see that Architecture-I is superior to others in the speed matter of the computation of independent multiplications, while Architecture-II is more suitable for computing dependent multiplications.

In Figs. 11 and 12 we illustrate the implementation of modular exponentiation with Architecture-I and Architecture-II acting as the main cores, respectively. In these figures, the controllers generate the controlling signals to collect the results of modular multiplications and to arrange the inputs of modular multiplications. According to an HSPICE simulation, the delay for Architecture-I is 1.6ns and the delay for Architecture-II is 2.3ns. To calculate exponentiations over GF(2155), Architecture-I can achieve 4 Mbit/s and Architecture-II can achieve 2.8 Mbit/s.

5 Conclusion

We have proposed two architectures for increasing the performance of multiplication over GF(2m). This was achieved by increasing the pipeline stage to shorten the clock cycle period, and by pairing off the cells to avoid the one-clock-cycle-gap problem. Among these two architectures, Architecture-I is suitable for calculating independent multiplications and Architecture-II is suitable for calculating dependent multiplications. Architecture-II has lower complexity in area-time. Two architectures are also proposed to compute exponentiations over GF(2m), based on our Architecture-I and Architecture-II, respectively. The architecture using Architecture-I as its main core can achieve 4 Mbit/s while the architecture using Architecture-II can achieve 2.8 Mbit/s.

6 References

2. DENNING, D.: ‘Cryptography and data security’ (Addison-Wesley, 1982)