Parallelizing Explicit Formula for Arithmetic in the Jacobian of Hyperelliptic Curves

Pradeep Kumar Mishra and Palash Sarkar

Cryptology Research Group,
Applied Statistics Unit,
Indian Statistical Institute,
203 B T Road, Kolkata-700108,
INDIA

Abstract. One of the recent thrust areas in research on hyperelliptic curve cryptography has been to obtain explicit formulae for performing arithmetic in the Jacobian of such curves. We continue this line of research by obtaining parallel versions of such formulae. Our first contribution is to develop a general methodology for obtaining parallel algorithm of any explicit formula. Any parallel algorithm obtained using our methodology is provably optimal in the number of multiplication rounds. We next apply this methodology to Lange's explicit formula for arithmetic in genus 2 hyperelliptic curve – both for the affine coordinate and inversion free arithmetic versions. Since encapsulated add-and-double algorithm is an important countermeasure against side channel attacks, we develop parallel algorithms for encapsulated add-and-double for both of Lange's versions of explicit formula. For the case of inversion free arithmetic, we present parallel algorithms using 4, 8 and 12 multipliers. All parallel algorithms described in this paper are optimal in the number of parallel rounds. One of the conclusions from our work is the fact that the parallel version of inversion free arithmetic is more efficient than the parallel version of arithmetic using affine coordinates.
Keywords : hyperelliptic curve cryptography, explicit formula, parallel algorithm, Jacobian, encapsulated add-and-double.

1 Introduction

Hyperelliptic curves present a rich source of abelian groups over which the discrete logarithm problem is believed to be difficult. Hence these groups can be used for implementation of various public key primitives.

The main operation in a hyperelliptic curve based primitive is scalar multiplication, which is the operation of computing mX, where m is an integer and X is a (reduced) divisor in the Jacobian of the curve. Any algorithm for scalar multiplication requires an efficient method of performing arithmetic in the Jacobian. This arithmetic essentially consists of two operations – addition and doubling of divisors.

The basic algorithm for performing arithmetic in the Jacobian of hyperelliptic curves is due to Cantor [1]. However, this algorithm is not sufficiently fast for practical implementation. There has been extensive research on algorithms for efficient arithmetic. The main technique is to obtain so called “explicit formula” for performing addition and doubling. These explicit formulae are themselves composed of addition, multiplication, squaring and inversion operations over the underlying finite field. Moreover, these formulae are specific to a particular genus. Thus there are separate formulae for genus 2 and genus 3 curves. See Table 1 in Section 2 for more details.

In this paper, we consider the problem of parallel execution of explicit formula. An explicit formula can contain quite a few field multiplications and squarings. (In certain cases, this can even be 50 or more.) On the other hand, the number of inversions is usually at most one or two. An explicit formula usually also contains many field additions; however, the cost of a field addition is significantly less than the cost of a field multiplication or inversion. Hence the dominant operation in an explicit formula is field multiplication.

On inspection of different explicit formulae appearing in the literature there appear to be groups of multiplication operations that can be executed in parallel. Clearly the ability to perform multiplications in parallel will improve the speed of execution of the algorithm. This gives rise to the following question: *Given an explicit formula, what is the best parallel algorithm for computing the formula?*

Our first contribution is to develop a general methodology for obtaining parallel version of any explicit formula. The methodology guarantees that the obtained parallel version requires the minimum number of rounds. The methodology can be applied to any explicit formula appearing in the literature. (There could also be other possible applications.)

The most efficient explicit formula for performing arithmetic in the Jacobian of genus 2 curve is given in [11, 12]. In [11], the affine coordinate representation of divisors is used and both addition and doubling involve a field inversion. On the other hand, in [12] the explicit formula is developed for inversion free arithmetic in the Jacobian.

Our second contribution is to apply our methodology to both [11] and [12]. For practical applications, it is necessary to consider resistance to side channel attacks. One important countermeasure is to perform a so-called encapsulated add-and-double algorithm (see [3, 6, 7] for details). We develop parallel versions of encapsulated add-and-double algorithm for both [11] and [12]. In many situations, the number of parallel multipliers available may be limited. To deal with such situations we present the encapsulated add-and-double algorithm using inversion free arithmetic using 4, 8 and 12 multipliers. For the affine version we have derived an algorithm using 8 multipliers. All of our algorithms are optimal parallel algorithms in the sense that no other parallel algorithm can perform the computation in lesser number of rounds.

Some of our results that we obtain are quite striking. For example, using 4 multipliers, we can complete the inversion free encapsulated add-and-double algorithm in 27 rounds and using 8 multipliers we can complete it in 14 rounds. The algorithm involves 108 multiplications. In the case of arithmetic using affine coordinates, the 8 multiplier algorithm will complete the computation in 11 rounds including an inversion round. Usually inversions are a few times costlier than multiplications, the actual figure being dependent upon exact implementation details. However, from our results it is clear that in general the parallel version of arithmetic using affine coordinates will be costlier than the parallel version of inversion free arithmetic.

2 Preliminaries of Hyperelliptic Curves

In this section, we give a brief overview of hyperelliptic curves. For details, readers can refer to [15]. Let K be a field and let $\overline{K}$ be the algebraic closure of K. A *hyperelliptic curve* C of genus g over K is an equation of the form $C : v^2 + h(u)v = f(u)$ where $h(u)$ in $K[u]$ is a polynomial of degree at most $g, f(u)$ in $K[u]$ is a monic polynomial of degree $2g + 1$, and there are no singular points (u, v) in $\overline{K} \times \overline{K}$. Unlike elliptic curves, the points on the hyperelliptic curve do not form a group. The additive group on which the cryptographic primitives are implemented is the divisor class group. Each element of this group is a *reduced divisor*. The group elements have a nice cannonical representation by means of two polynomials of small degree. The algorithms Koblitz [8] proposed for divisor addition and doubling are known as Cantor's algorithms.

Spallek [22] made the first attempt to compute divisor addition by explicit formula for genus 2 curves over fields of odd characteristic. Harley [5] improved the running time of the algorithm in [22]. Gaudry and Harley [4] observed that one can derive different explicit formula for divisor operations depending upon the weight of the divisors. In 2000, Nagao [19] proposed two algorithms; one for polynomial division without any inversion and another for extended gcd computation of polynomials requiring only one inversion. Both these algorithms can be applied to Cantor's algorithm to improve efficiency. Lange [10] generalised Harley's approach to curves over fields of even characteristic. Takahashi [23] and Miyamoto, Doi, Matsuo, Chao and Tsujii [17] achieved furthur speed-up using Montgomery's trick to reduce the number of inversions to 1. For genus 2 curves, the fastest version of explicit formula for inversion free arithmetic is given in [12] and the fastest version of explicit formula using affine coordinates is given in [11]. Lange has also proposed various co-ordinate systems and explicite formula for arithmetic of genus 2 curves over them. Interested readers can refer to [13]. For genus 3 curves Pelzl, Wollinger, Guajardo and Paar [20] have proposed explicit formula for performing arithmetic. For genus 4 curves, Pelzl, Wollinger and Paar have derived explicit formuae [21]. Curves of genus 5 and above are considered insecure for cryptographic use.

We summarise the complexity of various explicit formulae proposed in literature in Table 1. The cost generally correspond to the most general case. In the cost column, $[i], [m], [s]$ stand for the time taken by an inversion, a multiplication and a squaring in the underlying field respectively. The notation, $[m/s]$ stands for time of a square or multiplication. In the corresponding papers, multiplications and squarings have been treated to be of the same complexity.

Table 1. Complexity of Explicit Formulae

Genus	Name/Proposed in	Characteristic	Cost(Add)	Cost(Double)
Genus 2	Cantor [19]	All	$3[i] + 70[m/s]$	$3[i] + 76[m/s]$
	Nagao [19]	Odd	$1[i] + 55[m/s]$	$1[i] + 55[m/s]$
	Harley [5]	Odd	$2[i] + 27[m/s]$	$2[i] + 30[m/s]$
	Matsuo et al [14]	Odd	$2[i] + 25[m/s]$	$2[i] + 27[m/s]$
	Miyamoto et al [17]	Odd	$1[i] + 26[m/s]$	$1[i] + 27[m/s]$
	Takahashi [23]	Odd	$1[i] + 25[m/s]$	$1[i] + 29[m/s]$
	Lange [11]	All	$1[i] + 22[m] + 3[s]$	$1[i] + 22[m] + 5[s]$
	Lange [12]	All	$40[m] + 6[s]$	$47[m] + 4[s]$
Genus 3	Nagao [19]	Odd	$2[i] + 154[m/s]$	$2[i] + 146[m/s]$
	Pelzl et al [20]	All	$1[i] + 70[m] + 6[s]$	$1[i] + 61[m] + 10[s]$
Genus 4	Pelzl et al [21]	All	$2[i] + 160[m] + 4[s]$	$2[i] + 193[m] + 16[s]$

3 General Methodology for Parallelizing Explicit Formula

An explicit formula for performing doubling (resp. addition) in the Jacobian of a hyperelliptic curve is an algorithm which takes one (resp. two) reduced divisor(s) as input and produces a reduced divisor as output. Also the parameters of the curve are available to the algorithm. The algorithm proceeds by a sequence of elementary operations, where each operation is either a multiplication or an addition or an inversion over the underlying field. In general the formulae involve one inversion. If there is one inversion, the inversion operation can be neglected and the parallel version can be prepared without it. Later, it can be plugged in as a separate round at an appropriate place. The same is true if the formula contains more than one inversions. Hence, we can assume that the formula is inversion-free. The cost of a field multiplication (or squaring) is significantly more than the cost of a field addition and hence the number of field multiplications is the dominant factor determining the cost of the algorithm. On inspection of the different explicit formulae available in the literature, it appears that there are groups of multiplication operations which can be performed in parallel. The ability to perform several mulitplications in parallel can significantly improve the total computation time. So the key problem that we consider is the following: *Given an explicit formula, identify the groups of multiplication operations that can be performed in parallel.* In this section we develop a general methodology for solving this problem.

Let $\mathcal{F}$ be an explicit formula. Then $\mathcal{F}$ consists of mulitiplication and addition operations. Also several intermediate variables are involved. First we perform the following preprocessing on $\mathcal{F}$.

1. *Convert all multiplications to binary operation* : Operations which are expressed as a product of three or more variables are rewritten as a seqence of binary operations. For example, the operation $p_5 = p_1p_2p_3$ is rewritten as $p_4 = p_1p_2$ and $p_5 = p_3p_4$.

2. *Reduce multiplication depth* : Suppose we are required to perform the following sequence of operations: $p_3 = p_1^2 p_2;\ p_4 = p_3 p_2$. The straightforward way of converting to binary results in the following sequence of operations: $t_1 = p_1^2;\ p_3 = t_1 p_2;\ p_4 = p_3 p_2$. Note that the three operations *have* to be done sequentially one after another. On the other hand, suppose we perform the operations in the following manner: $\{t_1 = p_1^2;\ t_2 = p_2^2;\}\{p_3 = t_1 p_2;\ p_4 = t_1 t_2\}$. In this case, the operations within $\{\}$ can be performed in parallel and hence the computation can be completed in two parallel rounds. The total number of operations increases to 4, but the number of parallel rounds is less. We have performed such operation using inspection. We also note that it should be fruitful to consider algorithmic approach to this step.
3. *Eliminate reuse of variable names* : Consider the following sequence of operations:

$$q_1 = p_1 + p_2;\ q_2 = p_3;\ \ldots;\ q_1 = p_4 + p_5;\ \ldots$$

   In this case, at different points of the algorithm, the intermediate variable q_1 is used to store the values of both $p_1 + p_2$ and $p_4 + p_5$. During the process of devising the parallel algorithm we rename the variable q_1 storing the value of $p_4 + p_5$ by a unique new name. In the parallel algorithm we can again suitably rename it to avoid the overhead cost of initialising a new variable.
4. *Labeling process* : We assign unique labels to the addition and mulitplication operations and unique names to the intermediate variables.

Given a formula $\mathcal{F}$, we define a directed acyclic graph $G(\mathcal{F})$ in the following fashion.

- The nodes of $G(\mathcal{F})$ correspond to the arithmetic operations and variables of $\mathcal{F}$. Also there are nodes for the parameters of the input divisor(s) as well as for the parameters of the curve.
- The arcs are defined as follows: Suppose $\mathbf{id} : r = qp$ is a multiplication operation. The identifier **id** is the label assigned to this operation. Then the following arcs are present in $G(\mathcal{F})$: $(q, \mathbf{id}), (p, \mathbf{id})$ and $(\mathbf{id}, r)$. Similarly, the arcs for the addition operations are defined, with the only difference being the fact that the indegree of an addition node may be greater than two.

Proposition 1. *The following are true for the graph $G(\mathcal{F})$.*
1. The indegree of variable nodes corresponding to the parameters of the input divisors and the parameters of the curve is zero.
2. The indegree of any node corresponding to an intermediate variable is one.
3. The outdegree of any node corresponding to an addition or multiplication operation is one.

Note that the outdegree of nodes corresponding to variables can be greater than one. This happens when the variable is required as input to more than one arithmetic operation. Our aim is to identify the groups of multiplication operations that can be performed in parallel. For this purpose, we prepare another graph $G^*(\mathcal{F})$ from $G(\mathcal{F})$ in the following manner:

- The nodes of $G^*(\mathcal{F})$ are the nodes of $G(\mathcal{F})$ which correspond to multiplication operation.
- There is an arc $(\mathbf{id_1}, \mathbf{id_2})$ from node $\mathbf{id_1}$ to node $\mathbf{id_2}$ in $G^*(\mathcal{F})$ only if there is a path from $\mathbf{id_1}$ to $\mathbf{id_2}$ in $G(\mathcal{F})$ which does not pass through another multiplication node.

The graph $G^*(\mathcal{F})$ captures the ordering relation between the multiplication operations of $\mathcal{F}$. Thus, if there is an arc $(\mathbf{id_1}, \mathbf{id_2})$ in $G^*(\mathcal{F})$, then the operation $\mathbf{id_1}$ must be done before the operation $\mathbf{id_2}$. We now define a sequence of subgraphs of $G^*(\mathcal{F})$ and a sequence of subsets of nodes of $G^*(\mathcal{F})$ in the following manner.

- $G_1(\mathcal{F}) = G^*(\mathcal{F})$ and M_1 is the set of nodes of G_1 whose indegree is zero.
- For $i \geq 2$, G_i is the graph obtained from G_{i-1} by deleting the set M_{i-1} from G_{i-1} and M_i is the set of nodes of G_i whose indegree is zero.

Let r be the least positive integer such that G_{r+1} is the empty graph, i.e., on removing M_r from G_r, the resulting graph becomes empty.

Proposition 2. *The following statements hold for the graph $G^*(\mathcal{F})$.*
1. The sequence $M_1, \ldots, M_r$ forms a partition of the nodes of $G^*(\mathcal{F})$.
2. All the multiplications in any M_i can be performed in parallel.
3. There is a path in $G^*(\mathcal{F})$ from some vertex in M_1 to some vertex in M_r. Consequently, at least r parallel multiplication rounds are required to perform the computation of $\mathcal{F}$.

It is easy to obtain the sets M_i's from the graph $G^*(\mathcal{F})$ by a modification of the standard topological sort algorithm [2]. The sets M_i $(1 \leq i \leq r)$ represent only the multiplication operations of $\mathcal{F}$. To obtain a complete parallel algorithm, we have to organize the addition operations and take care of the intermediate variables. There may be some addition operations at the beginning of the formula. Since additions are to be performed sequentially, we can ignore these additions while deriving the parallelised formula, treating the sums they produce as inputs. Later, they can be plugged in at the beginning of the formula.

For $1 \leq i \leq r-1$, let A_i be the set of addition nodes which lie on a path from some node in M_i to some node in M_{i+1}. Further, let A_r be the set of addition nodes which lie on a path originating from some node in M_r. There may be more than one addition operation in a path from a node in M_i to a node in M_{i+1}. These additions have to be performed in a sequential manner. (Note that we are assuming that $\mathcal{F}$ starts with a set of multiplication operations and ends with a set of addition operations. It is easy to generalize to a more general form.)

Each multiplication and addition operation produces a value which is stored in an intermediate variable. We now describe the method of obtaining the set of intermediate variables required at each stage of computation. Let $I_1, \ldots, I_{2r}$ and $O_1, \ldots, O_{2r}$ be two sequences of subsets of nodes of $G(\mathcal{F})$, where each I_i and O_j contain nodes of $G(\mathcal{F})$ corresponding to variables. The parameters of the curve and the input divisor(s) are not included in any of the I_i and O_j's. These are assumed to be additionally present throughout the algorithm. For $1 \leq i \leq r$, these sets are defined as follows:

1. I_{2i-1} contains intermediate variables which are the inputs to the multiplication nodes in M_i.
2. I_{2i} contains intermediate variables which are the inputs to the addition nodes in A_i.
3. O_{2i-1} contains intermediate variables which are the outputs of the multiplication nodes in M_i.
4. O_{2i} contains intermediate variables which are the outputs of the addition nodes in A_i.

For $1 \le j \le 2r$, define

$$V_j = (\cup_{i=1}^{j} O_i) \cap (\cup_{i=j+1}^{2r} I_i). \tag{1}$$

If a variable x is in V_j, then it has been produced by some previous operation and will be required in some subsequent operation. We define the parallel version $\mathsf{par}(\mathcal{F})$ of $\mathcal{F}$ as a sequence of rounds

$$\mathsf{par}(\mathcal{F}) = (\mathcal{R}_1, \ldots, \mathcal{R}_r). \tag{2}$$

where $\mathcal{R}_i = (M_i, V_{2i-1}, A_i, V_{2i})$. In round i, the multiplications in M_i can be performed in parallel; the sets V_{2i-1} and V_{2i} are the sets of intermediate variables and A_i is the set of addition operations. Note that the addition operations are not meant to be performed in parallel. Indeed, in certain cases the addition operations in A_i have to be performed in a sequential manner. We define several parameters of $\mathsf{par}(\mathcal{F})$.

Definition 1. *Let* $\mathsf{par}(\mathcal{F}) = (\mathcal{R}_1, \ldots, \mathcal{R}_r)$*, be the* r*-round parallel version of the explicit formula* $\mathcal{F}$*. Then*
1. The total number of multiplications (including squarings) occuring in $\mathsf{par}(\mathcal{F})$ *will be denoted by* TM*.*
2. The multiplication width (MW) *of* $\mathsf{par}(\mathcal{F})$ *is defined to be* $\mathsf{MW} = \max_{1\le i\le r} |M_i|$*.*
3. The buffer width (BW) *of* $\mathsf{par}(\mathcal{F})$ *is defined to be* $\mathsf{BW} = \max_{1\le i\le 2r} |V_i|$*.*
4. A path from a node in M_1 *to a node in* M_r *is called a critical path in* $\mathsf{par}(\mathcal{F})$*.*
5. The value r *is the critical path length* (CPL) *of* $\mathsf{par}(\mathcal{F})$*.*

The parameter MW denotes the maximum number of multipliers that can operate in parallel. Using MW parallel multipliers $\mathcal{F}$ can be computed in r parallel rounds. The buffer width BW denotes the maximum number of variables that are required to be stored at any stage in the parallel algorithm.

3.1 Decreasing the Multiplication Width

The method described above yeilds a parallel algorithm $\mathsf{par}(\mathcal{F})$ for a given explicit formula $\mathcal{F}$. It also fixes the number of computational rounds r required to execute the algorithm using MW number of proessors. By definition, MW is the maximum number of multiplications taking place in a round. However, it may happen that in many rounds the actual number of multiplications is less than

MW. If we use MW multipliers, then some of the multipliers will be idle in such rounds. The most ideal scenario is $\mathsf{MW} \approx \lceil \mathsf{TM}/r \rceil$. However, such an ideal situation may not come about automatically. We next describe a method for making the distribution of the number of multiplication operations more uniform among various rounds.

We first prepare a *requirement table*. It is a table containing data about the intermediate variables created in the algorithm. For every variable it contains the name of the variables used in the expressions computing it, the latest round in which one of such variables is created and the earliest round in which the variable itself is used. For example, suppose an intermediate variable $v_x = v_y * v_z$ is computed in the j-th round. Of v_y and v_z, let v_z be the one which is computed later and in the i-th round. Let v_x be used earliest in the k-th round. Then in the requirement table we have an entry for v_x consisting of v_y, v_z, i, k. If both of v_x and v_y are input values then we may take $i = 0$. Note that we have $i < j < k$.

Now suppose, there are more than $\lceil \mathsf{TM}/r \rceil$ multiplications in the j-th round. Further suppose that for some j_1 $(i + 1 \leq j_1 \leq k - 1)$, the number of multiplications in the j_1^{th} round is less than $\lceil \mathsf{TM}/r \rceil$. Then we transfer the multiplication producing v_x to the j_1^{th} round and hence reduce the multiplication width of the j-th round. This change of position of the multiplication operation does not affect the correctness of the algorithm.

This procedure is applied as many times as possible to rounds which contain more than $\lceil \mathsf{TM}/r \rceil$ multiplications. As a result we obtain a parallel algorithm with a more uniform distribution of number of multiplication operations over the rounds and consequently reduces the value of MW.

3.2 Managing Buffer Width

The parameter BW provides the value of the maximum number of intermediate variables that is required to be stored at any point in the algorithm. This is an important parameter for applications where the amount of memory is limited. We justify that obtaining parallel version of an explicit formula does not substantially change the buffer width. Our argument is as follows.

First note that the total number of multiplications in the parallel version is roughly the same as the total number of multiplications in the original explicit formula. The only place where the number of multiplications increases is in the preprocessing step of reducing the multiplication depth. Moreover, the increase is only a few multiplications. The total number of addition operations remain the same in both sequential and parallel versions. Since the total numbers of multiplications and additions are roughly the same, the total number of intermediate variables also remains roughly the same.

Suppose that after round k in the execution of the parallel version, i intermediate variables have to be stored. Now consider a sequential execution of the explicit formula. Clearly, in the sequential execution, all operations upto round k has to be executed before any operation of round greater than k can be executed. The i intermediate variables that are required to be stored after round k

are required as inputs to operations in round greater than k. Hence these intermediate variables are also required to be stored in the sequential execution of the explicit formula.

4 Application to Lange's Explicit Formulae

In [11] and [12], Lange presented explicit formulae for addition and doubling in the Jacobian of genus 2 hyperelliptic curves. In fact, there are many special cases involved in these explicit formulae and our methodology can be applied to all the cases. But to be brief, we restrict our attention to the most general and frequent case only. The formulae in [11] uses an inversion each for addition and doubling while the formulae in [12] does not require any inversion.

We apply the methodology described in Section 3 separately to the formulae in [11] and [12]. In the case of addition, the inputs are two divisors D_1 and D_2 and in the case of doubling the input is only one divisor D_1. We use the following conventions.

- We assume that the curve parameters $h_2, h_1, h_0, f_4, f_3, f_2, f_1, f_0$ are available to the algorithm.
- We do not distinguish between squaring and multiplication.
- The labels for the arithmetic operations in the explicit formula for addition start with **A** and the labels for the arithmetic operations in the explicit formula for doubling start with **D**. The second letter of the label (**M** or **A**) denotes (m)ultiplication or (a)ddition over the underlying field. Thus **AM23** denotes the 23^{rd} multiplication in the explicit formula for addition.
- The intermediate variables for the explicit formula for addition are of the form p_i and the intermediate variables for the explicit formula for doubling are of the form q_j.
- In [11, 12], multiplications by curve constants are presented. However, during the total multiplication count, some of these operations are ignored, since for most practical applications the related curve constants will be 0 or 1. In this section, we include the multiplication by the curve parameters. In Section 5, we consider the situation where these are 0 or 1.
- The set of intermediate variables (V_i's) required at any stage is called the buffer state.

4.1 Inversion Free Arithmetic

In this section, we consider the result of application of the method of Section 3 to the inversion free formula for addition and doubling given in [12]. The details are presented in the Appendix. The details of addition formula is presented in Section A.1 and the details of the doubling formula is presented in Section A.2. We present a summary of the parameters of the parallel versions in Table 2. Based on Table 2 and Proposition 2(3), we obtain the following result.

Table 2. Parameters for parallel versions of explicit formula in [12].

	MW	BW	CPL	TM
Add	8	20	8	59
Double	11	15	8	65

Theorem 1. *Any parallel algorithm for executing either the explicit formula for addition or the explicit formula for doubling presented in [12] will require at least 8 parallel multiplication rounds. Consequently, the parallel algorithms presented in Sections A.1 and A.2 are optimal algorithms.*

4.2 Arithmetic Using Affine Coordinates

The most efficient explicit formula for arithmetic using affine coordinates has been presented in [11]. Here we consider the result of applying the methodology of Section 3 to this formula. Again due to lack of space we present the details full version of the paper. The parallel version of the addition formula is presented therein.

A summary of the results is presented in Table 3. We have the following

Table 3. Parameters for parallel versions of explicit formula in [11].

	MW	BW	CPL	TM
Add	6	12	7*	29*
Double	5	13	8*	34*

* Including one inversion

result about the parallel versions of the explicit formula in [11].

Theorem 2. *Any parallel algorithm for executing the explicit formula for addition (resp. doubling) presented in [11] will require at least 7 (resp. 8) parallel multiplication rounds. Consequently, the parallel algorithms presented in [16] are optimal algorithms.*

5 Encapsulated Addition and Doubling Algorithm

In this section, we address several issues required for actual implementation.

– The algorithms of Section A include multiplications by the parameters of the curve. However, we can assume that $h_2 \in \{0, 1\}$. If $h_2 \neq 0$, then by substituting $y = h_2^5 y^{'}$ and $x = h_2^2 x^{'}$ and dividing the resulting equation by h_2^{10}, we can make $h_2 = 1$. Also, if the underlying field is not of characteristic 5, we can assume that $f_4 = 0$. Otherwise, we can make it so by substituting $x^{'} = (x - f_4/5)$. In the algorithms presented below, we assume that $h_2 \in \{0, 1\}$ and $f_4 = 0$ and hence the corresponding multiplications are ignored. These decreases the total number of multiplications and hence also the number of parallel rounds. In most applications h_1, h_0 also are in $\{0, 1\}$. Hence efficiency in such situations can go up further. Thus all the operations in Section A of Appendix do not occur in the algorithms in this section.

- The usual add-and-double scalar multiplication algorithm is susceptible to side channel attacks. One of the main countermeasures is to perform both addition and doubling at each stage of scalar multiplication (see [3]). We call such an algorithm an encapsulated add-and-double algorithm. The parallel algorithms we present in this section are encapsulated add-and-double algorithms. All of them take as input two divisors D_1 and D_2 and produce as output $D_1 + D_2$ and $2D_1$.

5.1 Inversion Free Arithmetic

In this section, we consider parallel version of encapsulated add-and-double formula. We obtain the algorithms from the individual algorithms presented in Section A.1 and A.2.

First we note that the total number of multiplication operations for encapsulated add-and-double under the above mentioned conditions is 108. Since the value of MW for addition is 8 and for doubling is 11 and both have $\mathsf{CPL} = 8$, a total of 19 parallel finite field multipliers can complete encapsulated addition and doubling in 8 parallel rounds. However, 19 parallel finite field multipliers may be too costly. Hence we describe algorithms with 4, 8 and 12 parallel multipliers. (Note that an algorithm with two multipliers is easy to obtain – we assign one multiplier to perform addition and the other to perform doubling.)

Suppose the number of multipliers is m and the total number of operations is TM. Then at least $\lceil(\mathsf{TM}/m)\rceil$ parallel rounds are necessary. Any algorithm which performs the computation in these many rounds will be called a *best* algorithm. Our parallel algorithms with 4 and 8 multipliers are best algorithms. Further, our algorithm with 12 multipliers is optimal in the sense that no other parallel algorithm with 12 multipliers can complete the computation in less rounds.

The actual algorithms for performing inversion free arithmetic with 4 processors is presented in Table 5. Such tables for 8 and 12 processors are presented in the full version of the paper. This table only lists the multiplication and addition of field elements. Interested readers can access the full version of the paper at [16]. The labels in the table refer to the labels of operations in the algorithms in Section A.1 and A.2. We present a summary of the results in Table 2.

5.2 Affine Coordinates

An eight multiplier parallel version of explicit formula for encapsulated add-and-double is presented in the full version of the paper. In this case the total number of multiplications is 65. The eight multiplier algorithm requires 11 parallel rounds including an inversion round. On the other hand, the eight multiplier algorithm

Table 4. Summary of algorithms with varying number of processors for inversion free arithmetic of [12].

No of Multipliers	2	4	8	12
Number of rounds	54	27	14	10

for inversion free arithmetic requires only 14 multiplication rounds. Thus, in general the parallel version of inversion free arithmetic will be more efficient than the parallel version of arithmetic obtained from affine coordinates.

6 Conclusion

In this work, we have developed a general methodology for deriving parallel versions of any explicit formula for computation of divisor addition and doubling. We have followed the methods to derive the parallel version of the explicit formula given in [12] and [11]. We have considered encapsulated add-and-double algorithms to prevent side channel attacks. Moreover, we have described parallel algorithms with different number of processors.

It has been shown that for the inversion free arithmetic of [12] and with 4, 8 and 12 field multipliers an encapsulated add-and-double can be carried out in 27, 14 and 10 parallel rounds respectively. All these algorithms are optimal in the number of parallel rounds. In the case of arithmetic using affine coordinates [11], an eight multiplier algorithm can perform encapsulated add-and-double using 11 rounds including an inversion round. Since an inversion is usually several times costlier than a multiplication, in general the parallel version of inversion free arithmetic will be more efficient than the parallel version of arithmetic using affine coordinates.

We have applied our general methodology to explicit formula for genus 2 curves. The same methodology can also be applied to the explicit formula for genus 3 curves and to other explicit formulae appearing in the literature. Peforming these tasks will be future research problems.

References

1. D. G. Cantor. Computing in the Jacobian of a Hyperelliptic curve. In *Mathematics of Computation*, volume 48, pages 95-101, 1987.
2. T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, Cambridge, 1997.
3. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. *Proceedings of CHES 1999*, pp 292-302, 1999.
4. P. Gaudry and R. Harley Counting Points on Hyperelliptic Curves over Finite Fields. In *ANTS IV*, volume 1838 of LNCS; pp 297-312, Berlin, 2000, Springer-Verlag.
5. R. Harley. Fast Arithmetic on Genus 2 Curves. *Avaiable at http://cristal.inria.fr/ harley/hyper,2000.*

Table 5. Computation chart using four parallel multipliers for inversion free arithmetic of [12].

Rnd	Operation
1	AM01, AM02, AM03, AM04
2	AM05, AM06, AM07, AM08
	AA01, AA02, AA03, AA04
3	DM01, DM02, DM04, DM08
	DA01, DA02, DA03, DA04
4	DM09, AM09, AM10, AM11
	DA05, DA06, DA07, AA07, AA08, AA09
5	AM12, AM13, AM14, AM16
	AA05, AA06
6	DM12, DM13, DM14, DM15
	DA08
7	DM16, DM17, DM18, DM19
	DA09, DA10
8	DM20, DM22, AM17, AM18
	AA10, DA11, DA11, DA12, DA13
9	AM19, AM20, AM21, AM22
	AA12, AA13, AA14, AA15
10	DM23, DM24, DM25, DM26
	DA14, DA15, DA16, DA17, DA18, DA19
11	DM27, DM29, AM23, AM24
12	AM25, AM26, AM27, AM28
13	AM29, AM30, DM30, DM31
	AA16, AA17
14	DM32, DM33, DM34, DM35
	DA20, DA21

Rnd	Operation
15	AM31, AM32, AM33, AM34
	AA18, AA19
16	AM35, AM37, AM38, DM36
17	DM37, DM38, DM39, DM41
18	DM43, AM39, AM40, AM41
19	AM42, AM43, AM44, AM46
	AA20, AA21, AA22, AA23, AA24, AA25
20	DM44, DM45, DM46, DM47
21	DM48, DM49, DM50, AM47
	DA22, DA23, DA24, DA25
22	AM48, AM49, AM50, AM51
23	AM52, AM53, DM51, DM52
	AA26, AA27
24	DM53, DM54, DM55, DM56
25	DM57, AM54, AM55, AM56
	DA26, DA27, DA28
26	AM57, DM58, DM59, DM60
	AA28, AA29, AA30, AA31
27	DM62, DM63, DM65, DM66
	DA29, DA30, DA31, DA32, DA33, DA34

6. T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side-Channel Attacks Technical Report CORR 2002-03, University of Waterloo,2002. Available at http://www.cacr.math.uwaterloo.ca.
7. T. Izu, B. Möller and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant Against Side Channel Attacks. *Proceedings of Indocrypt 2002*, LNCS 2551, pp 296-313.
8. N. Koblitz. Hyperelliptic Cryptosystems. In *Journal of Cryptology*, 1: pages 139–150, 1989.
9. A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
10. T. Lange. Efficient Arithmetic on Hyperelliptic Curves. PhD thesis, Universität Gesamthochsschule Essen, 2001.
11. T. Lange. Efficient Arithmetic on Genus 2 Curves over Finite Fields via Explicit Formulae. Cryptology ePrint Archive, Report 2002/121, 2002. *http://eprint.iacr.org/*.
12. T. Lange. Inversion-free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. *http://eprint.iacr.org/*.
13. T. Lange. Weighted Co-ordinates on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/153, 2002. `http://eprint.iacr.org/`.
14. K. Matsuo, J. Chao and S. Tsujii. Fast Genus Two Hyperelliptic Curve Cryptosystems. In *ISEC2001, IEICE*,2001.
15. A. Menezes, Y. Wu, R. Zuccherato. An Elementary Introduction to Hyperelliptic Curves. Technical Report CORR 96-19, University of Waterloo(1996), Canada. Available at http://www.cacr.math.uwaterloo.ca.
16. P. K. Mishra and P. Sarkar Parallelizing Explicit Formula in the Jacobian of Hyperelliptic Curves (Full Version) Available at the Technical Report Section (Number 16) of `www://isical.ac.in/c̃rg`. Also available at IACR ePrint Archive, `http://eprint.iacr.org/`.
17. Y. Miyamoto, H. Doi, K. Matsuo, J. Chao and S. Tsujii. A fast addition algorithm for genus 2 hyperelliptic curves. In *Proc of SCIS2002, IEICE, Japan*, pp 497-502,2002, in Japanese.
18. P. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorisation. In *Math. Comp.*, vol 48, pp 243-264, 1987.
19. K. Nagao. Improving Group Law Algorithms for Jacobians of Hyperelliptic Curves. *ANTS IV*, LNCS 1838, Berlin 2000, Springer-Verlag.
20. J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves. Cryptology ePrint Archive, Report 2003/026, 2003. *http://eprint.iacr.org/*.
21. J. Pelzl, T. Wollinger and C. Paar. Low Cost Security: Explicit Formulae for Genus 4 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2003/097, 2003. *http://eprint.iacr.org/*.
22. A. M. Spallek. Kurven vom Geschletch 2 und irhe Anwendung in Public-Key-Kryptosystemen. PhD Thesis, Universität Gesamthochschule, Essen, 1994.
23. M. Takahashi. Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves. In *Proc of SCIS 2002*, ICICE, Japan, 2002, in Japanese.

A Details of Parallel Versions of Explicit Formula

The organisation of this section is as follows.

– Parallel version of the explicit formula for addition using inversion free arithmetic of [12] is presented in Section A.1.
– Parallel version of the explicit formula for doubling using inversion free arithmetic of [12] is presented in Section A.2.

Similar paralellised versions of addition and doubling algorithms for affine co-ordinates given in [11] have been derived using the methods presented in this paper and are available in the full version of the paper. Interested readers can find them at [16].

A.1 Addition Using Inversion Free Arithmetic

Algorithm
Input: Divisors $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$ and $D_2 = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$.
Output: Divisor $D_1 + D_2 = [U'_1, U'_0, V'_1, V'_0, Z']$
Initial buffer: $U_{11}, U_{10}, V_{11}, V_{10}, Z_1, U_{21}, U_{20}, V_{21}, V_{20}, Z_2$.

Round 1
AM01. $Z = Z_1 Z_2$; **AM02.** $\widetilde{U}_{21} = Z_1 U_{21}$; **AM03.** $\widetilde{U}_{20} = Z_1 U_{20}$;
AM04. $\widetilde{V}_{21} = Z_1 V_{21}$; **AM05.** $\widetilde{V}_{20} = Z_1 U_{20}$; **AM06.** $p_1 = U_{11} Z_2$;
AM07. $p_2 = U_{10} Z_2$; **AM08.** $p_3 = V_{11} Z_2$.
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_1, p_2, p_3$.
AA01. $p_4 = p_1 - \widetilde{U}_{21}$; **AA02.** $p_5 = \widetilde{U}_{20} - p_2$;
AA03. $p_6 = p_3 - \widetilde{V}_{21}$; **AA04.** $p_7 = Z_1 + U_{11}$.
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_3, p_4, p_5, p_6, p_{17}, p_7, Z$.

Round 2
AM09. $p_8 = U_{11} p_4$; **AM10.** $p_9 = Z_1 p_5$; **AM11.** $p_{10} = Z_1 p_4$;
AM12. $p_{11} = p_4^2$; **AM13.** $p_{12} = p_4 p_6$; **AM14.** $p_{13} = h_1 Z$;
AM15. $p_{14} = f_4 Z$; **AM16.** $p_{15} = V_{10} Z_2$
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_5, p_{17}, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}$.
AA05. $p_{16} = p_{15} - \widetilde{V}_{20}$; **AA06.** $p_{17} = p_{16} + p_6$;
AA07. $p_{18} = p_8 + p_9$; **AA08.** $p_{19} = p_{18} + p_{10}$;
AA09. $p_{20} = p_4 + \widetilde{U}_{21}$;
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_{17}, p_7, p_{12}, p_{13}, p_{14}, p_{18}, p_{19}, p_{20}$

Round 3
AM17. $p_{21} = p_5 p_{18}$; **AM18.** $p_{22} = p_{11} U_{10}$; **AM19.** $p_{23} = p_{19} p_{17}$;
AM20. $p_{24} = p_{18} p_{16}$ **AM21.** $p_{25} = p_{12} p_7$; **AM22.** $p_{26} = p_{12} U_{10}$;
Buffer: $Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, p_{15}, p_3, p_4, p_{13}, p_{14}, p_{20}, p_{21}, p_{22}, p_{23}, p_{24}, p_{25}, p_{26}$
AA10. $r = p_{21} + p_{22}$; **AA11.** $s_1 = p_{23} - p_{24} - p_{25}$;
AA12. $s_0 = p_{24} - p_{26}$; **AA13.** $p_{27} = \widetilde{U}_{21} + \widetilde{U}_{20}$;
AA14. $p_{28} = p_{13} + 2\widetilde{V}_{21}$; **AA15.** $p_{29} = p_4 + 2\widetilde{U}_{21} - p_{14}$;
Buffer:$Z, \widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, p_{15}, p_3, p_4, p_{20}, p_{27}, p_{28}, p_{29}$

Round 4

AM23. $R = Zr$; **AM24.** $s_0 = s_0 Z$; **AM25.** $s_3 = s_1 Z$;
AM26. $S = s_0 s_1$; **AM26.** $p_{30} = s_1 p_4$; **AM27.** $p_{31} = rp_{29}$;
AM28. $p_{32} = s_1 p_{28}$ **AM29.** $t = s_1 p_{20}$

Buffer: $\widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, R, s_3, S, t, p_{15}, p_3, p_4, p_{27}, p_{30}, p_{31}, p_{32}, p_{27}$

AA16. $p_{33} = s_0 - t$, **AA17.** $p_{34} = t - 2s_0$

Buffer:$\widetilde{U}_{21}, \widetilde{U}_{20}, \widetilde{V}_{21}, \widetilde{V}_{20}, r, s_1, s_0, R, s_3, S, p_{15}, p_3, p_4, p_{27}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}$

Round 5

AM30. $S_3 = s_3^2$; **AM31.** $\widetilde{R} = Rs_3$; **AM32.** $\widetilde{S} = s_3 s_1$;
AM33. $\widetilde{\widetilde{S}} = s_0 s_1$; **AM34.** $l_0 = S\widetilde{U}_{20}$; **AM35.** $p_{35} = h_2 p_{33}$;
AM36. $p_{36} = s_0^2$; **AM37.** $p_{37} = R^2$;

Buffer: $\widetilde{U}_{21}, \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{S}, S, p_{15}, p_3, p_{27}, p_{30}, p_{31}, p_{32}, p_{34}, p_{35}, p_{36}, p_{37}$

AA18. $p_{38} = \widetilde{S} + S$; **AA19.** $p_{39} = p_{35} + p_{32}$;

Buffer: $\widetilde{U}_{21}, \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{S}, p_{15}, p_3, p_{27}, p_{30}, p_{31}, p_{34}, p_{36}, p_{38}, p_{39}$

Round 6

AM38. $\widetilde{\widetilde{R}} = \widetilde{R}\widetilde{S}$; **AM39.** $l_2 = \widetilde{S}\widetilde{U}_{21}$; **AM40.** $p_{40} = p_{38} p_{27}$;
AM41. $p_{41} = p_{30} p_{34}$; **AM42.** $p_{42} = p_3 \widetilde{S}$; **AM43.** $p_{43} = Rp_{39}$;
AM44. $p_{44} = h_2 \widetilde{R}$; **AM45.** $p_{45} = p_{15} \widetilde{R}$;

Buffer: $\widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{\widetilde{R}}, p_{31}, p_{36}, p_{37}, p_{40}, p_{41}, p_{42}, p_{43}, p_{44}$

AA20. $l_1 = p_{40} - l_2 - l_0$; **AA21.** $l_2 = l_2 + \widetilde{\widetilde{S}}$;
AA22. $U_0' = p_{36} + p_{41} + p_{42} + p_{43} + p_{31}$;
AA23. $U_1' = 2\widetilde{\widetilde{S}} - p_{45} + p_{44} - p_{37}$;
AA24. $l_2 = l_2 - U_1'$; **AA25.** $p_{46} = U_0' - l_1$;

Buffer: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, l_2, l_0, S_3, \widetilde{R}, \widetilde{\widetilde{S}}, \widetilde{\widetilde{R}}, p_{46}$

Round 7

AM46. $p_{47} = U_0' l_2$; **AM47.** $p_{48} = S_3 l_0$; **AM48.** $p_{49} = U_1' l_2$;
AM49. $p_{50} = S_3 p_{46}$; **AM50.** $Z' = \widetilde{R} S_3$; **AM51.** $U_0' = \widetilde{R} U_0'$;
AM52. $U_1' = \widetilde{R} U_1'$;

Buffer state: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, \widetilde{\widetilde{R}}, p_{47}, p_{48}, p_{49}, p_{50}, Z'$

AA26. $p_{51} = p_{47} - p_{48}$; **AA27.** $p_{52} = p_{49} + p_{50}$;

Buffer: $U_0', U_1', \widetilde{V}_{21}, \widetilde{V}_{20}, \widetilde{\widetilde{R}}, p_{51}, p_{52}, Z'$

Round 8

AM53. $p_{53} = \widetilde{\widetilde{R}} \widetilde{V}_{20}$; **AM54.** $p_{54} = \widetilde{\widetilde{R}} \widetilde{V}_{21}$; **AM55.** $p_{55} = h_0 Z'$;
AM56. $p_{56} = h_1 Z'$; **AM57.** $p_{57} = h_2 U_0'$; **AM58.** $p_{58} = h_2 U_1'$;

Buffer state: $U_0', U_1', p_{51}, p_{52}, p_{53}, p_{54}, p_{55}, p_{55}, p_{56}, p_{57}, p_{58}, Z'$

AA28. $p_{59} = p_{51} - p_{53} - p_{55}$; **AA29.** $p_{60} = p_{52} - p_{54} - p_{56}$;
AA30. $V_0' = p_{57} + p_{59}$; **AA31.** $V_1' = p_{58} + p_{60}$;

Buffer state: $U_0^{'}, U_1^{'}, V_0^{'}, V_1^{'}, Z^{'}$

A.2 Doubling Using Inversion Free Arithmetic

Algorithm
Input: Divisors $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$.
Output: Divisor $2D_1 = [U_1^{''}, U_0^{''}, V_1^{''}, V_0^{''}, Z^{''}]$.
Initial Buffer: $U_{11}, U_{10}, V_{11}, V_{10}, Z_1$.

<u>**Round 1**</u>
DM01. $q_0 = Z_1^2$; **DM02.** $q_1 = h_1 Z_1$; **DM03.** $q_2 = h_2 U_{11}$;
DM04. $q_3 = h_0 Z_1$; **DM05.** $q_4 = h_2 U_{10}$; **DM06.** $q_5 = f_4 U_{11}$;
DM07. $q_6 = h_2 V_{11}$; **DM08.** $q_7 = f_2 Z_1$; **DM09.** $q_8 = V_{11} h_1$;
DM10. $q_9 = V_{10} h_2$; **DM11.** $q_{10} = f_4 U_{10}$;
Buffer: $q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}$
DA01. $\widetilde{V}_1 = q_1 + 2V_{11} - q_2$; **DA02.** $\widetilde{V}_0 = q_3 + 2V_{10} - q_4$;
DA03. $q_{11} = 2U_{10}$; **DA04.** $inv_1 = -\widetilde{V}_1$; **DA05.** $q_{12} = q_7 - q_8 - q_9 - 2q_{10}$;
DA06. $q_{13} = 2q_{11} + q_{10} + q_6$; **DA07.** $q_{14} = q_{11} + 2q_7 + q_6$;
Buffer: $inv_1, \widetilde{V}_1, \widetilde{V}_0, q_0, q_{14}, q_{11} q_{12}, q_{13}$

<u>**Round 2**</u>
DM12. $q_{15} = V_{11}^2$; **DM13.** $q_{16} = U_{11}^2$; **DM14.** $q_{17} = \widetilde{V}_0 Z_1$;
DM15. $q_{18} = U_{11} \widetilde{V}_1$; **DM16.** $q_{19} = \widetilde{V}_1^2$; **DM17.** $q_{20} = f_3 q_0$;
DM18. $q_{21} = q_{12} Z_1$; **DM19.** $q_{22} = q_{13} Z_1$; **DM20.** $q_{23} = q_{14} Z_1$;
DM21. $q_{24} = h_2 U_{11}$; **DM22.** $q_{25} = h_1 Z_1$;
Buffer:$inv_1, \widetilde{V}_1, \widetilde{V}_0, q_0, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}, q_{21}, q_{22}, q_{23}, q_{24}, q_{25}$
DA08. $q_{26} = q_{17} q_{18}$; **DA09.** $q_{27} = q_{20} + q_{16}$; **DA10.** $q_{28} = q_{22} - q_{27}$;
DA11. $k_1 = 2q_{16} + q_{27} - q_{23}$; **DA12.** $q_{29} = q_{21} - q_{15}$;
DA13. $q_{30} = 2V_{10} - q_{24} + q_{25}$;
Buffer:$inv_1, \widetilde{V}_0, k_1, q_0, q_{19}, q_{26}, q_{27}, q_{28}, q_{29}, q_{30}$

<u>**Round 3**</u>
DM23. $q_{31} = \widetilde{V}_0 q_{26}$; **DM24.** $q_{32} = q_{19} U_{10}$; **DM25.** $q_{33} = U_{11} q_{28}$;
DM26. $q_{34} = Z_1 q_{29}$; **DM27.** $q_{35} = k_1 inv_1$; **DM28.** $q_{36} = f_4 Z_1$;
DM29. $q_{37} = Z_1 U_{10}$;
Buffer:$inv_1, k_1, q_0, q_{26}, q_{31}, q_{32}, q_{37}, q_{30}, q_{33}, q_{34}, q_{35}, q_{36}$
DA14. $r = q_{31} + q_{32}$; **DA15.** $k_0 = q_{33} + q_{34}$; **DA16.** $q_{38} = k_0 + k_1$;
DA17. $q_{39} = inv_1 + q_{26}$; **DA18.** $q_{40} = 1 + U_{11}$;
DA19. $q_{41} = 2U_{11} - q_{36}$;
Buffer:$q_0, r, k_0, q_{26}, q_{37}, q_{30}, q_{35}, q_{36}, q_{38}, q_{39}, q_{40}, q_{41}$

<u>**Round 4**</u>
DM30. $R = q_0 r$; **DM31.** $q_{42} = q_{38} q_{39}$; **DM32.** $q_{43} = q_{35} q_{40}$;
DM33. $q_{44} = q_{35} q_{37}$; **DM34.** $q_{45} = k_0 q_{26}$; **DM35.** $q_{46} = r q_{41}$;
Buffer:$R, q_{30}, q_{45}, q_{36}, q_{42}, q_{43}, q_{44}, q_{46}$

DA20. $s_3 = q_{42} - q_{45} - q_{43}$; **DA21.** $s_0 = q_{45} - q_{44}$;
Buffer: $R, s_0, s_3, q_{30}, q_{46}$

Round 5
DM36. $q_{47} = R^2$; **DM37.** $q_{48} = s_0 s_3$; **DM38.** $s_1 = s_3 Z_1$;
DM39. $S_0 = s_0^2$; **DM40.** $t = h_2 s_0$; **DM41.** $q_{49} = q_{30} s_3$;
DM42. $q_{50} = h_2 R$; **DM43.** $q_{51} = Z_1 q_{46}$;
Buffer: $S_0, t, s_1, q_{47}, q_{48}, q_{49}, q_{51}, q_{50}$
Addition phase
No addition required at this step.
Buffer: Same as above.

Round 6
DM44. $\widetilde{R} = R s_1$; **DM45.** $S_1 = s_1^2$; **DM46.** $q_{52} = s_1 s_3$;
DM47. $S = q_{48} Z_1$; **DM48.** $l_0 = U_{10} q_{48}$; **DM49.** $q_{53} = R q_{49}$;
DM50. $q_{54} = q_{50} s_1$;
Buffer: $\widetilde{R}, S_1, S, S_0, t, l_0, q_{47}, q_{48}, q_{52}, q_{53}, q_{51}, q_{54}$
DA22. $q_{55} = U_{11} + U_{10}$; **DA23.** $q_{56} = q_{48} + q_{52}$;
DA24. $U_0^{''} = S_0 + q_{53} + t + q_{51}$;
DA25. $U_1^{''} = 2S + q_{54} - q_{47}$;
Buffer:$U_0^{''}, U_1^{''}, l_0, S_1, \widetilde{R}, q_{55}, q_{52}, q_{56}$

Round 7
DM51. $\widetilde{\widetilde{R}} = \widetilde{R} q_{52}$; **DM52.** $q_{57} = q_{56} q_{55}$; **DM53.** $q_{58} = S_1 l_0$;
DM54. $Z^{''} = S_1 \widetilde{R}$; **DM55.** $q_{59} = \widetilde{R} U_1^{''}$ **DM56.** $q_{60} = \widetilde{R} U_0^{''}$;
DM57. $l_2 = U_{11} s_1$;
Buffer: $U_0^{''}, U_1^{''}, Z^{''}, \widetilde{\widetilde{R}}, S_1, l_0, l_1, l_2, q_{57}, q_{58}, q_{59}, q_{60}$
DA26. $l_1 = q_{57} - l_2 - l_0$;
DA27. $l_2 = l_2 + S - U_1^{''}$; **DA28.** $q_{61} = U_0^{''} - l_1$;
Buffer:$U_0^{''}, U_1^{''}, Z^{''}, \widetilde{\widetilde{R}}, S_1, l_2, q_{58}, q_{59}, q_{60}, q_{61}$

Round 8
DM58. $q_{62} = U_0^{''} l_2$; **DM59.** $q_{63} = U_1^{''} l_2$; **DM60.** $q_{64} = S_1 q_{61}$;
DM61. $q_{65} = h_2 q_{60}$; **DM62.** $q_{66} = \widetilde{\widetilde{R}} V_{10}$; **DM63.** $q_{67} = h_0 Z^{''}$;
DM64. $q_{68} = h_2 q_{59}$; **DM65.** $q_{69} = \widetilde{\widetilde{R}} V_{11}$; **DM66.** $q_{70} = h_1 Z^{''}$;
Buffer: $Z^{''}, q_{58}, q_{59}, q_{60}, q_{62}, q_{63}, q_{64}, q_{65}, q_{66}, q_{67}, q_{68}, q_{69}, q_{70}$
DA29. $q_{71} = q_{62} + q_{58}$; **DA30.** $q_{72} = q_{63} + q_{64}$;
DA31. $U_0^{''} = q_{60}$; **DA32.** $U_1^{''} = q_{59}$;
DA33. $V_0^{''} = q_{71} + q_{65} - q_{66} - q_{67}$;
DA34. $V_1^{''} = q_{72} + q_{68} - q_{69} - q_{70}$;
Buffer:$U_0^{''}, U_1^{''}, Z^{''}, V_0^{''}, V_1^{''}$