Efficient Techniques for High-Speed Elliptic Curve Cryptography

Patrick Longa, and Catherine Gebotys

Department of Electrical and Computer Engineering,
University of Waterloo, Canada,
{plonga,cgebotys}@uwaterloo.ca

Abstract. In this paper, a thorough bottom-up optimization process (field, point and scalar arithmetic) is used to speed up the computation of elliptic curve point multiplication and report new speed records on modern x86-64 based processors. Our different implementations include elliptic curves using Jacobian coordinates, extended Twisted Edwards coordinates and the recently proposed Galbraith-Lin-Scott (GLS) method. Compared to state-of-the-art implementations on identical platforms the proposed techniques provide up to 30% speed improvements. Additionally, compared to the best previous published results on similar platforms improvements up to 31% are observed. This research is crucial for advancing high speed cryptography on new emerging processor architectures.

Keywords: Elliptic curve cryptosystem, point multiplication, point operation, field arithmetic, incomplete reduction, branch misprediction, data dependence, field arithmetic scheduling, software implementation.

1 Introduction

Elliptic curve point multiplication, defined as $[k]P$, where P is a point with order r on an elliptic curve $E(\mathbb{F}_p)$ and $k \in [1, r-1]$ is an integer, is the central and most time-consuming operation in Elliptic Curve Cryptography (ECC). Hence, its efficient realization on commodity processors, such as the new generation based on the x86-64 ISA, has gained increasing importance in recent years.

In this work, we combine several efficient techniques at the different computational levels of point multiplication to achieve significant speed improvements on x86-64 based CPUs:

- At the field arithmetic, code scheduling on hand-written assembly modules is carefully tuned for high performance field operations. Furthermore, optimal combination of well-known techniques such as incomplete reduction (IR) [21] and elimination of conditional branches is performed.
- At the point arithmetic, the cost of explicit formulas is reduced further by minimizing the number of additions/subtractions and small constants and maximizing the use of operations exploiting IR. Also, we study the negative effect of (true) data dependencies between "close" field operations and

propose *three* techniques to reduce their effect: field arithmetic scheduling, merging of point operations and merging of field operations.
– At the scalar arithmetic, we discuss our choice of recoding method and precomputation scheme and describe their efficient implementation.

Our implementations are carried out on elliptic curves using Jacobian and (extended) Twisted Edwards coordinates [13]. We also present results when applying the GLS method [8] that exploits an efficiently computable endomorphism to speed up the point multiplication over a quadratic extension field.

By efficiently combining the aforementioned techniques and other optimizations, we are able to compute a 256-bit point multiplication for the case of Jacobian and (extended) Twisted Edwards coordinates in only 337000 and 281000 cycles, respectively, on one core of an Intel Core 2 Duo processor. Compared to the previous results of 468000 and 362000 cycles (respect.) by Hisil et al. [14], our results achieve an improvement of about 28% and 22% (respect.). In the case of the GLS method, for Jacobian and (extended) Twisted Edwards coordinates, we compute one point multiplication in about 252000 and 229000 cycles (respect.) on the same processor, which compared to the best previous results by Galbraith et al. [7, 8] (326000 and 293000 cycles, respect.) translate to improvements of about 23% and 22%, respectively.

Our implementations use the well-known MIRACL library by M. Scott [20], which contains an extensive set of cryptographic functions that simplified the development/optimization process of our crypto routines. Our programs, based on M. Scott's software, are faster due to several improvements discussed in this paper. We greatly thank M. Scott for making his software freely available for educational purposes.

Although our programs are portable to any x86-64 based CPU, in this work we present test results on *three* processors: 2.66GHz Intel Core 2 Duo E6750, 2.83GHz Intel Xeon E5440 and 2.6GHz AMD Opteron 252.

Our work is organized as follows. In Section 2, we briefly introduce ECC over prime fields and the GLS method, and summarize the most relevant features of x86-64 based processors. In Sections 3, 4 and 5 we describe the different techniques employed for the speed-up of point multiplication at the field, point and scalar arithmetic levels. In Section 6, we discuss how our optimizations apply to implementations using GLS. Finally, in Section 7, we present our timings for point multiplication and compare them to the best previous results.

2 Preliminaries

For a background in elliptic curves, the reader is referred to [12]. In this work, we consider the standard elliptic curve equation E: $y^2 = x^3 + ax + b$ (also known as short Weierstrass equation) over a prime field $\mathbb{F}_p$, where $a, b \in \mathbb{F}_p$.

Representation of points using (x, y) is known as affine coordinates ($\mathcal{A}$). It is common practice to replace this representation with projective coordinates since affine coordinates are expensive over prime fields due to costly field inversions. In this work, we use Jacobian coordinates ($\mathcal{J}$), where each projective point

$(X : Y : Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$, $Z \neq 0$. The negative of $(X : Y : Z)$ is $(X : -Y : Z)$, and $(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z) : \lambda \in \mathbb{F}_p^*\}$.

The central operation, namely point multiplication (denoted by $[k]P$, for a point $P \in E(\mathbb{F}_p)$), is traditionally carried out through a series of point doublings and additions using some algorithm such as double-and-add. More efficiently, a doubling followed by another doubling can be computed as $\mathcal{J} \leftarrow 2\mathcal{J}$ and every doubling followed by an addition can utilize the new doubling-addition by [15, 19] to compute $\mathcal{J} \leftarrow 2\mathcal{J}+\mathcal{A}$ or $\mathcal{J} \leftarrow 2\mathcal{J}+\mathcal{J}$. All these formulas can also be found in our database of state-of-the-art formulas using Jacobian coord. [16].

Different curve forms exhibiting faster group arithmetic have been studied during the last few years. A good example is given by Twisted Edwards. This curve form, proposed in [2], is a generalization of Edwards curves [3] and has the equation $ax^2+y^2 = 1+dx^2y^2$, where $a, d \in \mathbb{F}_p$ are distinct nonzero elements. For this case, each triplet $(X : Y : Z)$ corresponds to the affine point $(X/Z, Y/Z)$, $Z \neq 0$, in homogeneous projective coordinates (denoted by $\mathcal{E}$). Later, Hisil et al. [13] introduced an extended system (called extended Twisted Edwards coord.; denoted by $\mathcal{E}^e$), where each point $(X : Y : Z : T)$ corresponds to $(X/Z, Y/Z, 1, T/Z)$ in affine, $T = XY/Z$ and $(X : Y : Z : T) = \{(\lambda X, \lambda Y, \lambda Z, \lambda T) : \lambda \in \mathbb{F}_p^*\}$.

Hisil et al. [13] also suggest the map $(x, y) \mapsto (x/\sqrt{-a}, y)$ to convert the previous curve to $-x^2+y^2 = 1+d'x^2y^2$, where $d' = -d/a$, allowing further reductions in the cost of point operations. For the point multiplication, they ultimately propose to compute a doubling followed by an addition as $\mathcal{E}^e \leftarrow 2\mathcal{E}$ and $\mathcal{E} \leftarrow \mathcal{E}^e+\mathcal{E}^e$ or $\mathcal{E} \leftarrow \mathcal{E}^e+\mathcal{A}$ (which can be unified into a doubling-addition operation with the form $\mathcal{E} \leftarrow (2\mathcal{E})^e+\mathcal{E}^e$ or $\mathcal{E} \leftarrow (2\mathcal{E})^e+\mathcal{A}$), and the remaining doublings as $\mathcal{E} \leftarrow 2\mathcal{E}$.

In Table 1, we have summarized the cost of formulas[1] using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$. Although variations to these formulas exist [16], these sometimes involve an increased number of "small" operations such as additions/subtractions. On some platforms, the extra cost may not be negligible. Formulas in Table 1 have been selected so that the *overall* cost is minimal on the targeted platforms. In Section 4, we apply some techniques to minimize the number of such "small" operations and, thus, to reduce the cost of point operations further.

Table 1. Cost of point operations on Weierstrass and Twisted Edwards curves.

Point Operation	Coord.	Weierstrass ($a = -3$)	Coord.	Twisted Edw. ($a = -1$)
Doubling	$\mathcal{J} \leftarrow 2\mathcal{J}$	4M+4S	$\mathcal{E} \leftarrow 2\mathcal{E}$	4M+3S
Mixed addition	$\mathcal{J} \leftarrow \mathcal{J}+\mathcal{A}$	8M+3S	$\mathcal{E} \leftarrow \mathcal{E}^e+\mathcal{A}$	7M
General addition	$\mathcal{J} \rightarrow \mathcal{J}+\mathcal{J}$	11M+3S (1)	$\mathcal{E} \leftarrow \mathcal{E}^e+\mathcal{E}^e$	8M
Mixed doubling-addition	$\mathcal{J} \leftarrow 2\mathcal{J}+\mathcal{A}$	13M+5S	$\mathcal{E} \leftarrow (2\mathcal{E})^e+\mathcal{A}$	11M+3S
General doubling-addition	$\mathcal{J} \rightarrow 2\mathcal{J}+\mathcal{J}$	16M+5S (1)	$\mathcal{E} \leftarrow (2\mathcal{E})^e+\mathcal{E}^e$	12M+3S

(1) Using cached values.

[1] Field operations: M = multiplication, S = squaring, Add = addition, Sub = subtraction, Mul$x$ = multiplication by x, Div$x$ = division by x, Neg = negation.

A recent method to improve the computation of point multiplication was proposed by Galbraith et al. [8], in which the computation is performed on a quadratic twist of a curve E over $\mathbb{F}_{p^2}$ with an efficiently computable homomorphism $\psi(x, y) \rightarrow (\alpha x, \alpha y)$, $\psi(P) = \lambda P$. Then, following [9], $[k]P$ can be computed as a multiple point multiplication with the form $[k_0]P + [k_1](\lambda P)$, where k_0 and k_1 have approximately half the bitlength of k. See [7, 8] for complete details.

In this work, we present two "traditional" implementations (on Weierstrass and Twisted Edwards curves) and another two using the GLS method (again, one per curve). For the traditional case (and to be competitive with other implementations in the literature), we have written the underlying field arithmetic over $\mathbb{F}_p$ using assembly language. On the other hand, for the GLS method we reuse the efficient modules for $\mathbb{F}_{p^2}$ field arithmetic provided with MIRACL.

For $\mathbb{F}_p$, we consider for maximal speed-up a pseudo-Mersenne prime with the form $p = 2^m - c$, where $m = n.w$ on an w-bit platform, $n \in \mathbb{Z}^+$, and c is a "small" integer (i.e., $c < 2^w$). These primes are highly efficient for performing modular reduction and support other optimizations such as elimination of conditional branches. Similarly, for the GLS method, field arithmetic over $\mathbb{F}_{p^2}$ provided by MIRACL considers a Mersenne prime $p = 2^t - 1$ (i.e., t is prime).

For a more in-depth treatment of the techniques exploited in our implementations, the reader is referred to the extended paper version [18].

2.1 The x86-64 based Processor Family

Modern CPUs from the desktop and server classes have decisively adopted the 64-bit x86 ISA (a.k.a. x86-64). This new instruction set expands general-purpose registers (GPRs) from 32 to 64 bits, allows arithmetic and logical operations on 64-bit integers and increments the number of GPRs, among other enhancements.

It seems that the move to 64 bits, with the inclusion of a powerful 64-bit integer multiplier, favors prime fields. Although the analysis becomes complex and processor dependent, our tests on the targeted processors suggest that SSE2 and its extensions seem not to be advantageous by themselves for the $\mathbb{F}_p$ arithmetic. This is probably due to the lack of carry handling and the fact that SSE2 multipliers can perform vector-multiplication with operands up to 32 bits only [11]. However, this outcome could change with improved SIMD extensions.

Another relevant feature of modern CPUs is their highly pipelined architectures. For instance, experiments by [6] suggest that Core 2 Duo and AMD architectures have pipelines with 15 and 12 stages, respectively. Although sophisticated branch prediction techniques exist, it is expected that the "random" nature of crypto computations, specifically of modular reduction, causes expensive mispredictions that force the pipeline to flush. In this work, we present experimental data quantifying the performance improvement obtained by eliminating branches in the field arithmetic (see Section 3).

Another direct consequence of highly pipelined architectures is that data dependencies between "close" instructions may insert a high penalty. Data dependencies that are relevant to our application are read-after-write (RAW), which can be found between a considerable number of field operations when the result

of a previous operation is required as input by the next operation. Our tests show that, if field operations are not scheduled properly, RAW dependencies can cause the pipeline to stall for several cycles degrading the performance significantly. In this work, we propose several techniques that help to minimize this problem, enhancing the performance of point multiplication (see Section 4).

3 Optimizations at the Field Arithmetic Level

In this section, we discuss the algorithms and optimizations that were applied to modular operations. All tests described were performed on our assembly language module implementing the field arithmetic over $\mathbb{F}_p$.

3.1 Field Multiplication

Schoolbook and Comba are the methods of choice for performing this operation on general purpose processors (GPPs). Methods such as Karatsuba multiplication theoretically reduce the number of integer multiplications but increase the number of other (cheaper) operations, which are not inexpensive in our case.

In x86-64 based CPUs, integer multiplication is relatively expensive. For instance, on an Intel Core 2 Duo, 64-bit multiplications can be executed every 5 clock cycles in a dependence chain [5]. A strategy to reduce costs is to interleave other (cheaper) operations with integer multiplications to exploit the *instruction-level parallelism* (ILP) found in modern processors. Precisely, both schoolbook (also known as operand scanning) and Comba's method (also known as product scanning) exhibit this attractive feature. Both methods require n^2 w-bit multiplications when multiplying two n-digit numbers. However, we choose to implement Comba's method since it requires approximately $3n^2$ w-bit additions, whereas schoolbook requires $4n^2$ (see Section 5.3.1 of [4]).

3.2 Other "Cheaper" Operations

There are *two* key techniques that we exploit to reduce the cost of additions, subtractions, and divisions/multiplications by small constants:

Incomplete Reduction (IR). This technique was introduced by Yanik et al. [21]. Given two numbers in the range $[0, p-1]$, it consists of allowing the result of an operation to stay in the range $[0, 2^s-1]$ instead of executing a complete reduction, where $p < 2^s < 2p-1$, $s = n.w$, w is the wordlength (e.g., $w = 64$) and n is the number of words. If the modulus is a pseudo-Mersenne prime with form $2^m - c$ such that $m = s$ and $c < 2^w$, the method gets even more advantageous. For example, in the case of addition the result can be reduced by first discarding the carry bit in the most significant word and then adding the correction value c.

In Table 2, we summarize the cost of field operations and the gain in performance when exploiting IR. As can be seen, in our experiments using $p = 2^{256} - 189$ we obtain significant reductions in cost ranging from 20% to up to 41%.

Table 2. Cost (in cycles) of modular operations when using incomplete reduction (IR) against complete reduction (CR) ($p = 2^{256} - 189$).

Modular Operation	Core 2 Duo E6750			Opteron 252		
	IR	CR	Cost reduction (%)	IR	CR	Cost reduction (%)
Addition	20	25	20%	13	20	35%
Multiplication by 2	19	24	21%	10	17	41%
Multiplication by 3	28	43	35%	15	23	35%
Division by 2	20	25	20%	11	18	39%

It is important to note that, because multiplication and squaring accept inputs in the range $[0, 2^s - 1]$, an operation using IR can precede any of these two operations. Then it turns out that virtually all additions and multiplications/divisions by small constants can be implemented with IR in our software.

Elimination of Conditional Branches. Following the trend of other crypto implementations [10, 20] and to avoid the high cost of branch misprediction on highly pipelined processors, we have implemented field addition, subtraction and multiplication/division by small constants without conditional branches; cf. [18].

In Table 3, we present the difference in performance for several field operations. In our tests using the prime $p = 2^{256} - 189$, we observed cost reductions as high as 50%. Remarkably, it can be seen that the greatest performance gains are obtained for operations exploiting IR. In conclusion, elimination of conditional branches favors more strongly our implementations, which are based on IR.

Table 3. Cost (in cycles) of modular operations without conditional branches (w/o CB) against operations using conditional branches (with CB) ($p = 2^{256} - 189$).

Modular Operation	Core 2 Duo E6750			Opteron 252		
	w/o CB	with CB	Cost reduction (%)	w/o CB	with CB	Cost reduction (%)
Subtraction	21	37	43%	16	23	30%
Addition with IR	20	37	46%	13	21	38%
Addition	25	39	36%	20	23	13%
Mult. by 2 with IR	19	38	50%	10	19	47%
Multiplication by 2	24	38	37%	17	20	15%

Finally, Table 4 summarizes the cost of field operations optimized with the techniques discussed above and used in our implementations, and compare them with `mpFq` [10], a well-known and highly-efficient crypto library. Note that timings for `mpFq` are reported for Intel Core 2 Duo 6700 and AMD Opteron 250 [10], which have very similar architectures to those used for our tests. Although our modular operations and those from `mpFq` are based on a different modulus p, comparisons in Table 4 are useful to explain part of the performance improvement obtained by our implementations in comparison with the implementation of *curve25519* using `mpFq` (see comparisons in Section 7).

Table 4. Cost (in cycles) of modular operations.

Modular Operation	Intel Core 2 Duo		AMD Opteron	
	This work $p = 2^{256} - 189$	mpFq [10] $p = 2^{255} - 19$	This work $p = 2^{256} - 189$	mpFq [10] $p = 2^{255} - 19$
Addition	20 (1)	21	13 (1)	19
Subtraction	21	24	16	22
Multiplication by 2	19 (1)	N/A	10 (1)	N/A
Division by 2	20 (1)	N/A	11 (1)	N/A
Squaring	101	107	65	72
Multiplication	110	141	80	108

(1) Using incomplete reduction.

4 Optimizations at the Point Arithmetic Level

In this section, we describe our choice of point formulas and some techniques to reduce their costs further. Also, we analyze how to reduce the computing cost of point multiplication by minimizing the number of pipeline stalls caused by contiguous field operations holding (true) data dependencies.

4.1 Our Choice of Explicit Formulas

For our programs, we choose the execution patterns based on doublings and doubling-additions proposed by Longa [15] and Hisil et al. [13] for $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$, respectively (see Section 2). For $\mathcal{J}$, we take as starting points the doubling formula from pp. 90 of [12] that costs 4M+4S, and the doubling-addition formula (3.5), pp. 37 of [15], that costs 13M+5S (16M+5S in the general case [16]). For $\mathcal{E}/\mathcal{E}^e$, we choose the doubling formula on pp. 400 of [2] that costs 4M+3S and the (dedicated) doubling-(dedicated) addition formulas from pp. 332-333 of [13] which cost in total 11M+3S (12M+3S in the general case). The previous costs (which are minimal on the targeted platforms in terms of mults and squarings) are obtained by setting $a = -3$ on $\mathcal{J}$ and $a = -1$ on $\mathcal{E}/\mathcal{E}^e$ [13] and avoiding the S-M tradings. Moreover, following [13], we precalculate (X_2+Y_2), (X_2-Y_2), $2Z_2$ and $2T_2$ to save *two* Adds and *two* Mul2 in the (dedicated) addition formula.

4.2 Minimizing the Cost of Point Operations

Further cost reduction of point operations can be achieved by exploiting the equivalence relation of projective coordinates. Consider, for example, the doubling formula using $\mathcal{J}$ in pp. 90-91 of [12] that has an overall cost of 4M+4S+ 1Add+4Sub+2Mul2+1Mul3+1Div2. If we fix $\lambda = 2^{-1} \in \mathbb{F}_p^*$ that formula can be modified to the following

$$X_2 = A^2 - 2B, \quad Y_2 = A\,(B - X_2) - Y_1^4, \quad Z_2 = Y_1 Z_1 \tag{1}$$

where $A = 3(X_1 + Z_1^2)(X_1 - Z_1^2)/2$, $B = X_1 Y_1^2$. With formula (1), the operation count is reduced to 4M+4S+1Add$_{\mathrm{IR}}$+5Sub+1Mul3$_{\mathrm{IR}}$+1Div2$_{\mathrm{IR}}$ (where

$operation_{\mathrm{IR}}$ represents an operation using incomplete reduction), replacing *two* multiplications by 2 with *one* subtraction and allowing the optimal use of incomplete reductions (every addition and multiplication/division by constants precedes a multiplication or squaring).

Additionally, depending on the relative cost of additions and subtractions (and the feasibility of using efficient "fused" subtractions such as $a - 2b \pmod p$; see Section 4.3) one may "convert" additions to subtractions (or vice versa) by applying $\lambda = -1 \in \mathbb{F}_p^*$ to a given formula. Refer to Appendix A for the details of the revised formulas exploiting these techniques.

4.3 Minimizing the Effect of Data Dependencies

Next, we present *three* techniques that help to reduce the number of memory stalls caused by RAW dependencies between successive field operations. For the remainder (and abusing notation), we define as *contiguous data dependence* if the output of a field operation is required as input by the immediately following operation causing the pipeline to stall in certain processor architecture.

Scheduling of Field Operations. The simplest solution to eliminate contiguous data dependencies is to perform a careful scheduling of the field operations inside point formulas. However, there is no unique solution and finding the optimal "arrangement" could be quite difficult and compiler/platform dependent. Instead, we demonstrate that some effort minimizing the number of these dependencies increases the overall performance significantly.

We tested several field operation "arrangements" to observe the potential impact of scheduling field operations. We detail here a few of our tests with field multiplication on an Intel Core 2 Duo. For example, let us consider the operation sequences given in Table 5. As can be seen, Sequence 1 involves a series of "ideal" data-independent multiplications, where the output of a given operation is not an input to the immediately following operation. In this case, the execution reaches its maximal performance with approx. 110 cycles/multiplication (see Table 4). Contrarily, the second sequence is highly-dependent because each output is required as input in the following operation. This is the worst-case scenario with an average of 128 cycles/mult., which is about 14% less efficient than the "ideal" case. We also studied other possible arrangements such as Sequence 3, in which operands of Sequence 2 have been reordered. This slightly amortizes the impact of contiguous data dependencies, improving the performance to 125 cycles/mult.

Table 5. Various sequences of field operations with different levels of contiguous data dependence. `Mult(op`i`,op`j`,res`k`)` denotes the field operation `res`k`←op`i`*op`j.

Sequence 1	Sequence 2	Sequence 3
`> Mult(op1,op2,res1)`	`> Mult(op1,op2,res1)`	`> Mult(op1,op2,res1)`
`> Mult(op3,op4,res2)`	`> Mult(res1,op4,res2)`	`> Mult(op4,res1,res2)`
`> Mult(op5,op6,res3)`	`> Mult(res2,op6,res3)`	`> Mult(op6,res2,res3)`
`> Mult(op7,op8,res4)`	`> Mult(res3,op8,res4)`	`> Mult(op8,res3,res4)`

Similarly, we have also tested the effect of contiguous data dependencies on other field operations, and detected that the cost reduction obtained by switching from an execution with strong contiguous data dependence (worst-case scenario, Sequence 2) to an execution with no contiguous data dependencies (best-case scenario, Sequence 1) ranges from approx. 9% to up to 33% on an Intel Core 2 Duo.

Merging point operations. This technique complements and increases the gain obtained by scheduling field operations. As expected, in some cases it is not possible to eliminate all contiguous data dependencies in a point formula by simple rescheduling. A clever way to increase the chances of eliminating more of these dependencies is by "merging" successive point operations.

It appears natural to merge successive doublings or a doubling and an addition. For our implementations, we use wNAF with window size $w = 5$ to recode the scalar (see Section 5). Then, at least *five* successive doublings between additions are expected. An efficient solution is to merge *four* consecutive doublings in a separate function and merge each addition with the precedent doubling in another function. In this way, we have been able to minimize most contiguous data dependencies and improve the overall performance further. As a side-effect, the number of function calls to point formulas is also reduced dramatically.

Merging field operations. If certain field operations are merged (and there are enough registers available) one can directly avoid memory stalls caused by dependencies between the writing to memory of the result and its posterior reading in the following field operation. A positive side-effect of this approach is that memory accesses (and potential cache misses) are also minimized.

Some crypto libraries have already experimented with this approach to certain extent. For example, MIRACL includes a double subtraction operation that executes $a-b-c \pmod p$ and a multiplication by 3 executed as $a+a+a \pmod p$. However, in this work we have maximized the use of registers and included other combinations such as $a-2b \pmod p$ and the merging of $a-b \pmod p$ and $(a-b)-2c \pmod p$. We remark that this list is not exhaustive. Different platforms with more registers or different coordinate systems/underlying fields may enable a much wider range of merging options (for instance, see Section 6 for the merging options suggested for quadratic extension fields).

To illustrate the impact of scheduling field operations, merging point operations and merging field operations, we show in Table 6 the cost of a point doubling when using these techniques in comparison with a naïve implementation with a high number of dependencies.

Table 6. Cost (in cycles) of point doubling with different number of contiguous data dependencies (Jacobian coordinates, $p = 2^{256} - 189$).

Technique	# contiguous data-depend. per doubling	Core 2 Duo E6750		Opteron 252	
		Cost per doubling	Relative reduction (%)	Cost per doubling	Relative reduction (%)
"Unscheduled"	10	1115	–	786	–
Scheduled/merged	1.25	979	12%	726	8%

As shown in Table 6, by reducing the number of dependencies from *ten* to about *one* per doubling, minimizing function calls and reducing the number of memory reads/writes, we are able to reduce the cost of a doubling by 12% and 8% on Core 2 Duo and Opteron processors, respectively.

Following the strategies presented in this section, we have first minimized the cost of point operations (cf. §4.2) and then carefully scheduled (merged) field operations inside (merged) point operations so that memory stalls and memory accesses are minimized. See Appendix A for costs and scheduling details of most relevant point operations used in our implementations and discussed in §4.1.

5 Optimizations at the Scalar Arithmetic Level

In this section, we describe our choice of algorithms for the computation of point multiplication and precomputation.

For scalar recoding we use width-w Non-Adjacent Form (wNAF), which offers minimal nonzero density among signed binary representations for a given window width [1]. In particular, we use Alg. 3.35 of [12] for conversion from integer to wNAF representation. Although left-to-right conversion algorithms exist [1], which save memory and allow on-the-fly computation of point multiplication, they are not advantageous on the targeted CPUs. In fact, our tests show that converting the scalar to wNAF and then executing the point multiplication achieves higher performance than interleaving both stages. This could be explained by the fact that the latter approach "interrupts" the otherwise smooth flow of point multiplication by calling the conversion function at every iteration of the double-and-add algorithm.

For precomputation on $\mathcal{J}$, we have chosen a variant of the LM scheme [19] that does not require inversions (see Section 7.1 of [17]). This method achieves the lowest precomputing cost, given by $(5L+2)$M+$(2L+4)$S, where L represents the number of non-trivial points (note that we avoid here the S-M trading in the first doubling). On $\mathcal{E}/\mathcal{E}^e$ coordinates, we precompute points using the traditional sequence $P + 2P + \ldots + 2P$, adding $2P$ with general additions. Because precomputed points are left in projective coordinates no inversion is required and the cost is given by $(8L + 4)$M+2S. For both $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$, we have chosen a window with size $w = 5$ (i.e., precomputing $\{P, [3]P, \ldots, [15]P\}, L = 7$), which is optimal and slightly better than fractional windows using $L = 6$ or $L = 8$.

6 Implementation using GLS

For our implementations using GLS, we apply similar techniques to those described in Sections 4 and 5 for the elliptic curve arithmetic. As mentioned previously, we use the optimized assembly implementation of the field arithmetic over $\mathbb{F}_{p^2}$ by M. Scott [20]. This library exploits the "nice" Mersenne prime $2^{127} - 1$, which allows a very simple reduction step with no conditional branches.

Note that the field arithmetic over $\mathbb{F}_{p^2}$ in fact translates to a bunch of $\mathbb{F}_p$

operations, where p has 127 bits in our case. For instance, each multiplication using Karatsuba (as implemented in [20]) involves 3 $\mathbb{F}_p$ multiplications and 5 $\mathbb{F}_p$ additions/subtractions. Thus, the scheduling and merging of field operations described in Section 4.3 are first applied to this underlying layer over $\mathbb{F}_p$ and then extended to the upper layer over $\mathbb{F}_{p^2}$.

For the point arithmetic, we slightly modify formulas described in Section 4 and Appendix A since in this case these require a few extra multiplications with the twisted curve parameter μ (see Appendix B). For example, the (dedicated) addition using $\mathcal{E}/\mathcal{E}^e$ with cost 8M has to be replaced with a formula that costs 9M (discussed in pp. 332 of [13] as "9M+1D"). Moreover, field arithmetic over $\mathbb{F}_{p^2}$ enables a much richer opportunity for merging field operations. In our implementations, we include $a-2b\,(\mathrm{mod}p)$, $(a+a+a)/2\,(\mathrm{mod}p)$, $a+b-c\,(\mathrm{mod}p)$, the merging of $a+b\,(\mathrm{mod}\,p)$ and $a-b\,(\mathrm{mod}\,p)$, the merging of $a-b\,(\mathrm{mod}\,p)$ and $c-d\,(\mathrm{mod}p)$, and the merging of $a+a\,(\mathrm{mod}p)$ and $a+a+a\,(\mathrm{mod}p)$. For complete details about point formulas and their implementation for the GLS method, the reader is referred to Appendix B in the extended paper version [18].

For the multiple point multiplication $[k_0]P+[k_1](\lambda P)$, each of the two scalars k_0 and k_1 is converted using fractional wNAF, and then the evaluation stage is executed using interleaving (see Alg. 3.51 of [12]). Again, we remark that the separation of the conversion and evaluation stages yields better performance in our case. For precomputation on $\mathcal{J}$, we use the LM scheme (see Section 4 of [19]) that has minimal cost among methods using only *one* inversion, i.e., 1I+(9L+1)M+(2L+5)S (we avoid here the S-M trading in the first doubling). A fractional window with $L=6$ achieves the optimal performance in our case. Again, on $\mathcal{E}/\mathcal{E}^e$ we precompute points using general additions in the sequence $P+2P+\ldots+2P$. Precomputed points are better left in projective coordinates, in which case the cost is given by (9L+4)M+2S. In this case, an integral window $w=5$ (i.e., $L=7$) achieves optimal performance. As pointed out by [8], precomputing $\{P,[3]\psi(P),\ldots,[2L+1]\psi(P)\}$ can be done on-the-fly at low cost.

7 Implementation Results

In this section, we summarize the timings obtained by our "traditional" implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ (called *ted256189* and *jac256189*, respect.), and our implementations using GLS (called *ted1271gls* and *jac1271gls*, respect.), when running them on a single core of the targeted x86-64 based CPUs. The curves used in these implementations are described in detail in Appendix B. For verification of each implementation, the results of 10^4 point multiplications with "random" scalars were all validated using MIRACL. Several "random" point multiplications were also verified with Magma.

All the tested programs were compiled with gcc v4.4.1 on the Intel Core 2 Duo E6750 and with gcc v4.3.4 on the Intel Xeon E5440 and Opteron 252 processors. For measuring computing time, we follow [10] and use a method based on cycle counts. To obtain our timings, we ran each implementation 10^5 times with randomly generated scalars, averaged and approximated the results to

the nearest 1000 cycles. Table 7 summarizes our results, labeled as *ted1271gls*, *jac1271gls*, *ted256189* and *jac256189*. All costs include scalar conversion, the point multiplication computation (precomputation and evaluation stages) and the final normalization step to affine. Table 7 also shows the cycle counts that we obtained when running the implementations by M. Scott (displayed as *gls1271-ref4* and *gls1271-ref3* [20]) on exactly the same platforms. Finally, the last 5 rows of the table detail cycle counts of several state-of-the-art implementations as reported in the literature. However, these referenced results are used only to provide an approximate comparison since the processor platforms are not identical (though they use very similar processors).

Table 7. Cost (in cycles) of point multiplication.

Implementation	Coord.	Field Arithm.	Core 2 Duo E6750	Xeon E5440	Opteron 252
ted1271gls	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_{p^2}$, 127-bit	**229000**	**230000**	**211000**
jac1271gls	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	252000	255000	238000
ted256189	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	281000	289000	232000
jac256189	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	337000	343000	274000
gls1271-ref4 [20]	$\mathcal{E}^{\text{inv}}$	$\mathbb{F}_{p^2}$, 127-bit	295000	296000	295000
gls1271-ref3 [20]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	332000	332000	341000
gls1271-ref4 [7]	$\mathcal{E}^{\text{inv}}$	$\mathbb{F}_{p^2}$, 127-bit	293000 (1)	–	–
gls1271-ref3 [8]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	326000 (1)	–	–
curve25519 [10]	Montgomery	$\mathbb{F}_p$, 255-bit	386000 (2)	–	307000 (4)
Hisil et al. [14]	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	362000 (3)	–	–
Hisil et al. [14]	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	468000 (3)	–	–

(1) On a 1.66GHz Intel Core 2 Duo. (2) On a 2.66GHz Intel Core 2 Duo E6700.
(3) On a 2.66GHz Intel Core 2 Duo E6550. (4) On a 2.4GHz AMD Opteron 250.

As can be seen in Table 7, our fastest implementation on the targeted platforms is *ted1271gls*, using $\mathcal{E}/\mathcal{E}^e$ with the GLS method. This implementation is about 22% faster than the previous record set by *gls1271-ref4* [7] on a slightly different processor (1.66GHZ Intel Core 2 Duo). A more precise comparison, however, would be between measurements on identical processor platforms. In this case, *ted1271gls* is approx. 22%, 22% and 28% faster than *gls1271-ref4* [20] on Intel Core 2 Duo E6750, Intel Xeon E5440 and AMD Opteron 252, respectively. Although [20] uses inverted Twisted Edwards coordinates ($\mathcal{E}^{\text{inv}}$), the improvement with the change of coordinates only explains a small fraction of the speed-up. Similarly, in the case of $\mathcal{J}$ combined with GLS, *jac1271gls* is about 23% faster than the record set by *gls1271-ref3* [8] on a 1.66GHZ Intel Core 2 Duo. When comparing cycle counts on identical processor platforms, *jac1271gls* is 24%, 23% and 30% faster than *gls1271-ref3* [20] on Intel Core 2 Duo E6750, Intel Xeon E5440 and AMD Opteron 252, respect. Our implementations are also significantly faster than the implementation of Bernstein's *curve25519* by Gaudry and Thomé [10]. For instance, *ted1271gls* is 41% faster than *curve25519* [10] on a 2.66GHz Intel Core 2 Duo.

If GLS is not considered, the fastest implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ are

ted256189 and *jac256189*, respectively. In this case, *ted256189* and *jac256189* are 22% and 28% faster than the previous best cycle counts due to Hisil et al. [14] using also $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$, respect., on a 2.66GHz Intel Core 2 Duo.

It is also interesting to note that the performance boost given by the GLS method depends on the characteristics of a given platform. For instance, *ted1271gls* and *jac1271gls* are about 19% and 25% faster than their "counterparts" over $\mathbb{F}_p$, namely *ted256189* and *jac256189*, respect., on a Core 2 Duo E6750. However, on the AMD Opteron processor the gap between the costs of field operations over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ is shorter. As consequence, on Opteron 252 *ted1271gls* and *jac1271gls* only achieve a reduction of approx. 9% and 13% with respect to *ted256189* and *jac256189*, respectively. For the record, *ted1271gls* achieves the best cycle count on an AMD Opteron with an advantage of about 31% over the best previous result in the literature, i.e., *curve25519* [10].

In summary, this paper has illustrated that a significant speed-up can be achieved using a combination of optimizing techniques applied to all levels of the ECC computation and adapted to the architectural features of modern processors. This research is crucial for advancing the state-of-the-art crypto implementations in present and future platforms. Also, although our implementations (in their current form) only compute $[k]P$ where k and P vary, several of the optimizations discussed in this work are generic and can be easily adapted to speed up other implementations using a fixed point P, digital signatures and different coordinate systems/curve forms/underlying fields.

Acknowledgments. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada. We would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Centres of Excellence (OCE) for partially supporting this work. We would also like to thank Mike Scott, Hiren Patel and the reviewers for their useful comments.

References

1. Avanzi, R.: A Note on the Signed Sliding Window Integer Recoding and its Left-to-Right Analogue. In: SAC 2004. LNCS, vol. 3357, pp. 130–143. Springer, Heidelberg (2005)
2. Bernstein, D., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards Curves. In: AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008)
3. Edwards, H.: A Normal Form for Elliptic Curves. In: Bulletin of the American Mathematical Society, vol. 44, pp. 393–422 (2007)
4. Erdem, S.S., Yanik, T., Koç, Ç.K.: Fast Finite Field Multiplication. In: Ç.K. Koç (ed.) Cryptographic Engineering, Chapter 5. Springer (2009)
5. Fog, A.: Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs (2009), `http://www.agner.org/optimize/#manuals`, accessed on January 2010
6. Fog, A.: The Microarchitecture of Intel, AMD and VIA CPUs (2009), `http://www.agner.org/optimize/#manuals`, accessed on January 2010

7. Galbraith, S., Lin, X., Scott, M.: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. Cryptology ePrint Archive, Report 2008/194 (2008)
8. Galbraith, S., Lin, X., Scott, M.: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. In: EUROCRYPT 2009. LNCS, Vol. 5479, pp. 518-535. Springer, Heidelberg (2009)
9. Gallant, R., Lambert, R., Vanstone, S.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In: CRYPTO 2001. LNCS, vol. 2139, pp. 190-200, Springer, Heidelberg (2001)
10. Gaudry, P., Thomé, E.: The mpFq Library and Implementing Curve-Based Key Exchanges. In: SPEED 2007, pp. 49-64 (2007)
11. Hankerson, D., Menezes, A., Scott, M.: Software Implementation of Pairings. In: M. Joye and G. Neven (eds.) Identity-Based Cryptography, Chapter 12. IOS Press (2009)
12. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag (2004)
13. Hisil, H., Wong, K., Carter, G., Dawson, E.: Twisted Edwards Curves Revisited. In: ASIACRYPT 2008. LNCS, vol. 5350, pp. 326-343. Springer, Heidelberg (2008)
14. Hisil, H., Wong, K., Carter, G., Dawson, E.: Jacobi Quartic Curves Revisited. Cryptology ePrint Archive, Report 2009/312 (2009)
15. Longa, P.: Accelerating the Scalar Multiplication on Elliptic Curve Cryptosystems over Prime Fields. Master's Thesis, University of Ottawa (2007), `http://patricklonga.bravehost.com/publications.html#thesis`
16. Longa, P.: ECC Point Arithmetic Formulae (EPAF) (2008), `http://patricklonga.bravehost.com/jacobian.html`
17. Longa, P., Gebotys, C.: Setting Speed Records with the (Fractional) Multibase Non-Adjacent Form Method for Efficient Elliptic Curve Scalar Multiplication. CACR technical report, CACR 2008-06 (2008)
18. Longa, P., Gebotys, C.: Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors (2010), `http://patricklonga.bravehost.com/publications.html` (to appear)
19. Longa, P., Miri, A.: New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields. In: PKC 2008. LNCS, vol. 4939, pp. 229-247. Springer, Heidelberg (2008)
20. Scott, M.: MIRACL - Multiprecision Integer and Rational Arithmetic C/C++ Library (1988-2007), `ftp://ftp.computing.dcu.ie/pub/crypto/miracl.zip`
21. Yanik, T., Savaş, E., Koç, Ç.K.: Incomplete Reduction in Modular Arithmetic. In: IEE Proc. of Computers and Digital Techniques, vol. 149(2), pp. 46-52 (2002)

A Point Operations using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates

The Maple scripts below verify formulas used in our "traditional" implementations. Revised formulas for the GLS method can be found in the extended paper version [18]. Note that field operations have been carefully merged and scheduled to reduce pipeline stalls and memory reads/writes. Also, several point formulas are merged to reduce further the number of pipeline stalls and minimize the number of function calls. Temporary registers are denoted by t_i, DblSub represents $a-2b \pmod p$ and SubDblSub merges $a-b \pmod p$ and $(a-b)-2c \pmod p$.

Underlined operations are merged.

```
# Weierstrass curve (for verification):
x1:=X1/Z1^2; y1:=Y1/Z1^3; x2:=X2/Z2^2; y2:=Y2/Z2^3; ZZ2:=Z2^2; ZZZ2:=Z2^3; a:=-3;
x3:=((3*x1^2+a)/(2*y1))^2-2*x1; y3:=((3*x1^2+a)/(2*y1))*(x1-x3)-y1;
x4:=((y1-y2)/(x1-x2))^2-x2-x1; y4:=((y1-y2)/(x1-x2))*(x2-x4)-y2;
x5:=((y1-y4)/(x1-x4))^2-x4-x1; y5:=((y1-y4)/(x1-x4))*(x4-x5)-y4;
```

DBL, $\mathcal{J} \leftarrow 2\mathcal{J}$: $(X_{out}, Y_{out}, Z_{out}) \leftarrow 2(X_1, Y_1, Z_1)$. Cost = 4M+4S+3Sub+1DblSub+1Add$_{\text{IR}}$+1Mul3$_{\text{IR}}$+1Div2$_{\text{IR}}$; 5 contiguous data depend.

```
# In practice, Xout,Yout,Zout reuse the registers X1,Y1,Z1 for all cases below.
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t0:=3*t4; t5:=X1*t3; t4:=t1*t0; t0:=t3^2; t1:=t4/2;
t3:=t1^2; Zout:=Y1*Z1; Xout:=t3-2*t5; t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
simplify([x3-Xout/Zout^2]), simplify([y3-Yout/Zout^3]); # Check
```

4DBL, $\mathcal{J} \leftarrow 8\mathcal{J}$: $(X_{out}, Y_{out}, Z_{out}) \leftarrow 8(X_1, Y_1, Z_1)$. Cost = 4*(4M+4S+3Sub+1DblSub+1Add$_{\text{IR}}$+1Mul3$_{\text{IR}}$+1Div2$_{\text{IR}}$); 1.25 contiguous data depend./doubling

```
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t2:=3*t4; t5:=X1*t3; t4:=t1*t2; t0:=t3^2; t1:=t4/2;
Zout:=Y1*Z1; t3:=t1^2; t4:=Z1^2; Xout:=t3-2*t5; t3:=t5-Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-
t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2; t4:=t1*t2; t5:=Xout*t3; t1:=t4/2; t0:=t3^2; t3:=t1^2;
Zout:=Yout*Zout; Xout:=t3-2*t5; t4:=Zout^2; t3:=t5-Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4;
Yout:=t5-t0; t1:=3*t4; t3:=Yout^2;  t4:=t1*t2;  t5:=Xout*t3; t1:=t4/2;  t0:=t3^2;  t3:=t1^2;
Zout:=Yout*Zout; Xout:=t3-2*t5; t4:=Zout^2; t3:=t5-Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4;
Yout:=t5-t0; t1:=3*t4; t3:=Yout^2; t4:=t1*t2;  t5:=Xout*t3;  t1:=t4/2;  t0:=t3^2;  t3:=t1^2;
Zout:=Yout*Zout; Xout:=t3-2*t5; t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
```

mDBLADD, $\mathcal{J} \leftarrow 2\mathcal{J} + \mathcal{A}$: $(X_{out}, Y_{out}, Z_{out}) \leftarrow 2(X_1, Y_1, Z_1) + (x_2, y_2)$.
Cost = 13M+5S+7Sub+2DblSub+1Add$_{\text{IR}}$+1Mul2$_{\text{IR}}$; 5 contiguous data depend.

```
t5:=Z1^2; t6:=Z1*t5; t4:=x2*t5; t5:=y2*t6; t1:=t4-X1; t2:=t5-Y1; t4:=t2^2; t6:=t1^2; t5:=t6*
X1; t0:=t1*t6; t3:=t4-2*t5; t4:=Z1*t1; t3:=t3-t5; t6:=t0*Y1; t3:=t3-t0; t1:=2*t6; Zout:=t4*t3;
t4:=t2*t3; t0:=t3^2; t1:=t1+t4; t4:=t0*t5; t7:=t1^2; t5:=t0*t3; Xout:=t7-2*t4; Xout:=Xout-t5;
t3:=Xout-t4; t0:=t5*t6; t4:=t1*t3; Yout:=t4-t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

DBLADD, $\mathcal{J} \leftarrow 2\mathcal{J}+\mathcal{J}$: $(X_{out}, Y_{out}, Z_{out}) \leftarrow 2(X_1, Y_1, Z_1)+(X_2, Y_2, Z_2, Z_2^2, Z_2^3)$.
Cost = 16M+5S+7Sub+2DblSub+1Add$_{\text{IR}}$+1Mul2$_{\text{IR}}$; 3 contiguous data depend.

```
t0:=X1*ZZ2; t5:=Z1^2; t7:=Y1*ZZZ2; t4:=X2*t5; t6:=t5*Z1; t1:=t4-t0; t5:=Y2*t6; t6:=t1^2; t2:=
t5-t7; t4:=t2^2; t5:=t6*t0; t0:=t1*t6; t3:=t4-2*t5; t6:=Z1*t1; t3:=t3-t5; t4:=Z2*t6; t3:=t3-t0;
t6:=t7*t0; Zout:=t4*t3; t4:=t2*t3; t1:=2*t6; t0:=t3^2; t1:=t1+t4; t4:=t0*t5; t7:=t1^2; t5:=t0*t3;
Xout:=t7-2*t4; Xout:=Xout-t5; t3:=Xout-t4; t0:=t5*t6; t4:=t1*t3; Yout:=t4-t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

```
# Twisted Edwards curve (for verification):
x1:=X1/Z1; y1:=Y1/Z1; x2:=X2/Z2; y2:=Y2/Z2; T2:=X2*Y2/Z2; a:=-1;
x3:=(2*x1*y1)/(y1^2+a*x1^2); y3:=(y1^2-a*x1^2)/(2-y1^2-a*x1^2);
x4:=(x3*y3+x2*y2)/(y3*y2+a*x3*x2); y4:=(x3*y3-x2*y2)/(x3*y2-y3*x2);
```

DBL, $\mathcal{E} \leftarrow 2\mathcal{E}$: $(X_{out}, Y_{out}, Z_{out}) \leftarrow 2(X_1, Y_1, Z_1)$. Cost = 4M+3S+1SubDblSub+1Add$_{\text{IR}}$+1Mul2$_{\text{IR}}$+1Neg; no contiguous data dependencies

```
t1:=2*X1; t2:=X1^2; t4:=Y1^2; t3:=Z1^2; Xout:=t2+t4; t4:=t4-t2; t3:=t4-2*t3; t2:=t1*Y1; Yout:=
-t4; Zout:=t4*t3; Yout:=Yout*Xout; Xout:=t3*t2;
simplify([x3-Xout/Zout]), simplify([y3-Yout/Zout]); # Check
# Iterate this code n times to obtain nDBL with cost n(4M+3S+1SubDblSub+1AddIR+1Mul2IR+1Neg)
```

Merged DBL–ADD, $\mathcal{E} \leftarrow (2\mathcal{E})^e+\mathcal{E}^e : (X_{out}, Y_{out}, Z_{out}) \leftarrow 2(X_1, Y_1, Z_1)+((X_2+Y_2), (X_2 - Y_2), 2Z_2, 2T_2)$. Cost = 12M+3S+3Sub+1SubDblSub+4Add$_{\text{IR}}$+1Mul2$_{\text{IR}}$; no contiguous data dependencies

```
# If Z2=1 (Merged DBL-mADD), t5:=(2*Z2)*t6 is replaced by t5:=2*t6 and the number of multiplies
reduces to 11M at the expense of one extra Mul2
t1:=2*X1; t5:=X1^2; t7:=Y1^2; t6:=Z1^2; Xout:=t5+t7; t7:=t7-t5; t6:=t7-2*t6; t5:=t1*Y1; t8:=t7*
Xout; t0:=t7*t6; t7:=t6*t5; t6:=Xout*t5; Xout:=t7+t8; t1:=t7-t8; t7:=(2*T2)*t0; t5:=(2*Z2)*t6;
t0:=(X2-Y2)*t1; t1:=t5+t7; t6:=(X2+Y2)*Xout; Xout:=t5-t7; t7:=t0-t6; t0:=t0+t6; Xout:=Xout*t7;
Yout:=t1*t0; Zout:=t0*t7;
simplify([x4-Xout/Zout]), simplify([y4-Yout/Zout]); # Check
```

B The Curves

The curves below provide approximately 128-bit level of security and were found by using a modified version of the Schoof's algorithm provided with MIRACL.

- *Jac256189* uses the Weierstrass curve $E_w : y^2 = x^3 - 3x + B$ over $\mathbb{F}_p$ with $\mathcal{J}$, where $p = 2^{256} - 189$, $B =$ 0xfd63c3319814da55e88e9328e96273c483dca6cc84 df53ec8d91b1b3e0237064 and $\#E_w(\mathbb{F}_p) = 10r$ (r is a 253-bit prime).

- *Ted256189* uses the Twisted Edwards curve $E_{tedw}: -x^2 + y^2 = 1 + 358\,x^2y^2$ over $\mathbb{F}_p$ with $\mathcal{E}/\mathcal{E}^e$, where $p = 2^{256} - 189$ and $\#E_{tedw}(\mathbb{F}_p) = 4r$ (r is a 255-bit prime).

- *Jac1271gls* uses the quadratic twist $E'_{w-gls}: y^2 = x^3 - 3\mu x + 44\mu$ of the Weierstrass curve $E_{w-gls}(\mathbb{F}_{p^2})$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square, $E_{w-gls}/\mathbb{F}_p$: $y^2 = x^3 - 3x + 44$ and $p = 2^{127} - 1$. In this case, $\#E'_{w-gls}(\mathbb{F}_{p^2})$ is a 254-bit prime. The same curve is also used in [8].

- *Ted1271gls* uses the quadratic twist $E'_{tedw-gls} : -\mu x^2 + y^2 = 1 + 109\mu\, x^2y^2$ of the Twisted Edwards curve $E_{tedw-gls}(\mathbb{F}_{p^2})$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square, $E_{tedw-gls}/\mathbb{F}_p : -x^2 + y^2 = 1 + 109\,x^2y^2$ and $p = 2^{127} - 1$. In this case, $\#E'_{tedw-gls}(\mathbb{F}_{p^2}) = 4r$ where r is a 252-bit prime.