# A Cryptographic Processor for Arbitrary Elliptic Curves over GF($2^m$)

Hans Eberle, Nils Gura, Sheueling Chang-Shantz
Sun Microsystems Laboratories
{Nils.Gura, Hans.Eberle, Sheueling.Chang}@sun.com

**Abstract**

*We describe a cryptographic processor for Elliptic Curve Cryptography (ECC). ECC is evolving as an attractive alternative to other public-key schemes such as RSA by offering the smallest key size and the highest strength per bit. The processor performs point multiplication for elliptic curves over binary polynomial fields $GF(2^m)$. In contrast to other designs that only support one curve at a time, our processor is capable of handling arbitrary curves without requiring reconfiguration. More specifically, it can handle both named curves as standardized by NIST as well as any other generic curves up to a field degree of 255. Efficient support for arbitrary curves is particularly important for the targeted server applications that need to handle requests for secure connections generated by a multitude of heterogeneous client devices. Such requests may specify curves which are infrequently used or not even known at implementation time.*

*Our processor implements 256-bit modular multiplication, division, addition and squaring. The multiplier constitutes the core function as it executes the bulk of the point multiplication algorithm. We present a novel digit-serial modular multiplier that uses a hybrid architecture to perform the reduction operation needed to reduce the multiplication result: Hardwired logic is used for fast reduction of named curves and the multiplier circuit is reused for reduction of generic curves. The performance of our FPGA-based prototype, running at a clock frequency of 66.4 MHz, is 6955 point multiplications per second for named curves over $GF(2^{163})$ and 3308 point multiplications per second for generic curves over $GF(2^{163})$.*

## 1 Introduction

In this paper, we describe the architecture and implementation of a processor for Elliptic Curve Cryptography (ECC). ECC is a public-key cryptosystem that is rapidly evolving as an attractive alternative to other schemes such as RSA by offering the smallest key size and the highest strength per bit. For example, a 163-bit ECC key offers the same security strength as a 1024-bit RSA key. The ratio of key sizes is going to favor ECC even more as larger keys are adopted. As a result of the smaller key size, some cryptographic operations such as signing can be executed much faster with ECC [7]. For this reason, ECC is most interesting to emerging wireless technologies that use Internet-ready mobile phones, PDAs, smart cards and sensor networks. Since these types of client devices have modest to little compute resources, the computational efficiency makes ECC a good match.

Several standards have been created to specify the use of ECC. The US government has adopted ECC for the Elliptic Curve Digital Signature Algorithm (ECDSA) and rec-

ommended a set of curves. For binary polynomial fields, the curves cover key sizes of 163, 233, 283, 409 and 571 bit [15]. Additional curves for commercial use were recommended by the Standards for Efficient Cryptography Group (SECG) [3]. Also, efforts are underway to include ECC into security protocols such as OpenSSL - SSL/TLS is today's dominant Internet security protocol [12, 6].

Terminating secure connections on the server side not only demands high computational power but also flexibility in responding to client devices that are limited in the set of cryptographic algorithms supported. As clients are often limited in processing power and memory capacity, they may be capable of supporting only a small number of curves. With respect to ECC, a client might possibly support a single curve only. To be able to establish a secure connection and, with it, provide service, a server, in turn, is required to be flexible enough to support any such curve requested by a client. While a server certainly needs to implement the standardized curves and the associated irreducible polynomials, it should further implement any other arbitrary curve and, thus, any arbitrary irreducible polynomial. In the following, we refer to the former as *named curves* and to the latter as *generic curves*. There are several reasons why generic curves need to be supported. The standards only recommend curves and, thus, new curves might emerge in the future. Furthermore, curves might be abandoned for security reasons and replaced by different ones that were not known at implementation time.

While previous work has targeted implementations optimized for specific curves, our design has the unique property that it provides optimized performance for multiple named curves and support for arbitrary generic curves. In a previous publication [9], we introduced a technique called partial reduction that allows for generic curve-independent implementations of ECC. While our previous publication only described a firmware implementation of this technique, we are now introducing a novel digit-serial multiplier that implements modular multiplication in hardware for both named and generic curves thereby significantly improving performance for the latter.

## 2 Related Work

Hardware and firmware implementations of ECC point multiplication over different fields $GF(2^m)$ have been reported in numerous publications. A design for $GF((2^8-17)^{17})$, optimized for 8-bit processors, is described by Woodbury et al. in [16]. The implementation targets a SmartCard based on an Intel 8051 microcontroller. Orlando and Paar describe a programmable elliptic curve processor for reconfigurable logic in [13]. Different curves can be handled by parameterizing the hardware architecture and reconfiguring the logic. Bednara et al. [2] designed an FPGA-based ECC processor architecture that allows for using multiple squarers, adders and multipliers. Two prototypes were synthesized for $GF(2^{191})$. Agnew et al. [1] built an ASIC implementing ECC point multiplication for $GF(2^{155})$. Goodman and Chandrakasan [5] designed a generic public-key processor that executes modular operations on integer and binary polynomial fields. The data path can be reconfigured to support different field degrees. Point multiplication over binary polynomial fields is computed by a microcoded double-and-add algorithm. To our knowledge, this is the only implementation that supports $GF(2^m)$ for variable field degrees $m$. However, the architecture is optimized for low power consumption and its performance cannot be scaled to levels required by server-type applications. All other implementations described above

target either one or a small number of specific curves. That is, none of them can handle a curve that is not specified at implementation time without requiring the software to be modified or the hardware to be reconfigured.

## 3 ECC Processor Architecture

We chose a microprogrammable architecture for the ECC processor. The microprogram is stored in static memory and uploaded by the host at initialization time. Although the functionality of the ECC processor is fixed, controlling program execution by a microprogram rather than hardwired control logic provided an ideal platform for experimenting with different point multiplication algorithms.

### 3.1 Data Path

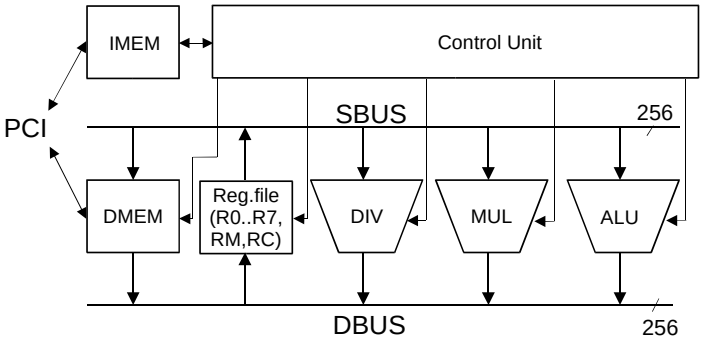


**Figure 1.** *Data Path and Control Unit.*

We decided on a bus structure for the data path to keep the design as flexible as possible. This design decision proved to be valuable as it allowed us to easily change the function units without affecting the communication infrastructure. Figure 1 shows the data path and the control unit. Dual-ported instruction and data memories IMEM and DMEM connect the ECC processor with the PCI bus of the host system. The internal data path is $n = 256$ bits wide, that is, the busses, the register file and memories are 256 bits wide and the function units operate on 256-bit operands. The data memory DMEM, the registers and the function units are connected by the busses SBUS and DBUS. The data memory DMEM has a capacity of 8 kBytes and stores parameters and variables. The register file contains eight general purpose registers R0-R7, a register RM that holds the irreducible polynomial and a register RC that specifies the field degree and the type of curve; more specifically, it specifies whether a named curve or a generic curve is to be processed. The function units include a modular divider (DIV), a modular multiplier (MUL), and a multifunction arithmetic and logic unit (ALU). The ALU provides addition, modular squaring, shift, and comparison functions.

### 3.2 Instruction Set

The ECC processor implements a load/store architecture. That is, memory can be accessed by load and store operations only, and all arithmetic instructions are limited to register operands. Instructions fall into three categories: Memory instructions, arithmetic instructions, and control instructions. All instructions have a fixed length of 16 bits. Table 1 contains the complete instruction set.

| Opcode | | Name | Semantics | Cycles |
|---|---|---|---|---|
| **Memory Instructions** | | | | |
| LD | DMEM,RD | Load | DMEM → RD | 3 |
| ST | RS,DMEM | Store | RS → DMEM | 3 |
| **Arithmetic Instructions** | | | | |
| DIV | RS0,RS1,RD | Divide | (RS1/RS0) mod M → RD | ≤ 2m+4 / ≤ 2n+4 |
| MUL | RS0,RS1,RD | Multiply | (RS0*RS1) mod M → RD | 7/8/10/12 |
| ADD | RS0,RS1,RD | Add | RS0+RS1 → RD<br>(RD==0) → EQ | 3 |
| SQR | RS,RD | Square | (RS*RS) mod M → RD<br>(RD==0) → EQ | 3 |
| SL | RS,RD | Shift Left | {RS[254..0],0} → RD<br>RS[255] → MZ<br>(RD==0) → EQ | 3 |
| **Control Instructions** | | | | |
| BMZ | ADDR | Branch | branch if MZ == 0 | 2 |
| BEQ | ADDR | Branch | branch if EQ == 1 | 4 |
| BNC | ADDR | Branch | branch if NC is set | 2 |
| JMP | ADDR | Jump | jump | 2 |
| NOP | | No Operation | no operation | 1 |
| END | | End | end program execution | |

**Table 1.** *Instruction Set.*

### 3.3 Control Unit

The control unit consists of the instruction memory IMEM that has a capacity of 1 kByte or 512 instructions and a finite state machine (FSM) that controls the data path according to the instructions fetched. The FSM can handle variable execution times as occur for MUL and DIV instructions. For the multiplier, the cycle count varies with the field degree $m$, and for the divider, the cycle count depends on both the field degree $m$ and the values of the operands.

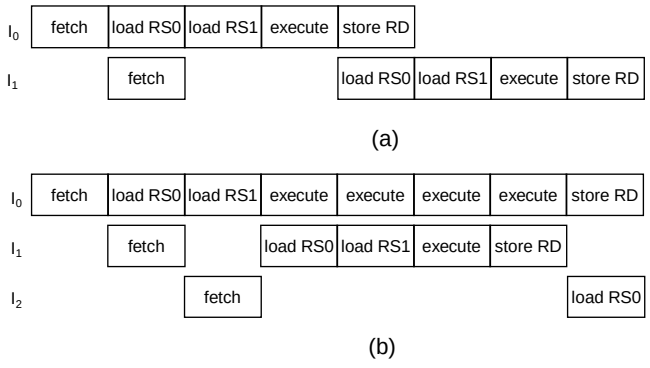


**Figure 2.** *Overlapped (a) and Parallel (b) Instruction Execution.*

Program execution times are further optimized by overlapping instruction execution and executing instructions in parallel. The control unit overlaps the execution of arithmetic instructions by prefetching the instruction as well as preloading the first source operand. This is illustrated in Figure 2a. Data dependencies are detected by the assembler and are considered programming errors. Often, these dependencies can be resolved by swapping the source operands. However, for SQR, SL, ST, or DIV, such a dependency cannot be removed as suggested and a NOP instruction needs to be inserted.

Parallel execution of instructions is implemented in that an ADD or SQR instruction can be executed in parallel to a MUL instruction if there are no data dependencies. This is shown in Figure 2b The choice of these particular instructions is motivated by an analysis of the program code for point multiplication. As will be shown in Table 4, the MUL instruction is the most frequently executed instruction and, in many instances, can be executed in parallel with either an ADD or a SQR instruction.

### 3.4 Function Units

The function units perform modular arithmetic operations on binary polynomials in standard basis representation. The ALU implements the two arithmetic instructions ADD and SQR and the logic instruction SL. ADD translates into a bit-wise XOR of the two source operands. SQR requires the insertion of zeroes between the bits of the source operand and the subsequent reduction of the so expanded source operand. A hardwired reduction circuit is used that can only handle named curves.

The ECC processor implements a modular divider based on an algorithm described by Chang-Shantz [4] that has similarities to Euclid's GCD algorithm. The multiplier constitutes the core of the data path and is described in detail in the next section.

## 4 Multiplier

As the performance analysis contained in Section 6 shows, more than half the number of cycles required to process a point multiplication are spent in the multiplier. For this reason, we optimized its performance as much as possible and spent a significant part of the chip resources on it.

We have implemented a number of digit-serial modular multiplier designs based on algorithms described by Song and Parhi in [14]. Our first design described in [8] used a least significant digit (LSD) first multiplier that could perform modular multiplication in hardware for named curves only. In addition, the multiplier could generate an unreduced product so that reduction for generic curves could be performed by microcode. With this implementation, generic curves were processed at a tenth of the throughput achieved for named curves. Here, we describe a novel multiplier design that performs modular multiplication for both types of curves in hardware thereby significantly improving the performance for generic curves. The new design is based on a most significant digit (MSD) first multiplier. We also considered the LSD first multiplier but, as we will explain later, found that pipelining for the MSD multiplier can be done more efficiently.

We will first describe an MSD first multiplier that works for named curves only, before we describe the final design that can handle both named and generic curves. The pseudo code looks as follows:

```
X[n-1..0] := x*t^(d*⌊(n−m)/d⌋); Y[n-1..0] := y*t^(d*⌊(n−m)/d⌋);
P[n+d-1..0] := 0; Z[n-1..0] := 0;
for i := 0 to ⌈m/d⌉-1 do
  P[n+d-1..0] := X[n-1..n-d] * Y[n-1..0];
  X[n-1..0] := shift_left(X[n-d-1..0],d);
  Z[n-1..0] := (shift_left(Z[n-1..0],d) + P[n+d-1..0]) mod M*t^(d*⌊(n−m)/d⌋);
end;
```

Figure 3a shows a block diagram of an MSD first multiplier for named curves of field degrees 163, 193, and 233. The following three computation steps are performed in parallel: (i) the MSD of $X$ is multiplied with $Y$; (ii) $X$ is shifted to the left by $d$ bits; (iii) $Z$ is shifted to the left by $d$ bits, added to $P$, and subsequently reduced. It takes $\lceil m/d \rceil + 1$ clock cycles to perform the modular multiplication, that is, the number of multiplication steps executed depends on $m$. This optimization requires that the registers $X$ and $Y$ are loaded with the operands shifted to the left by $d * \lfloor (n-m)/d \rfloor$ bits. In our implementation, we only support

a shift by $d$ bits. That is, for $n = 256$ and $d = 64$, the modular multiplication takes five clock cycles for $m > 192$ and four clock cycles for $m \leq 192$.

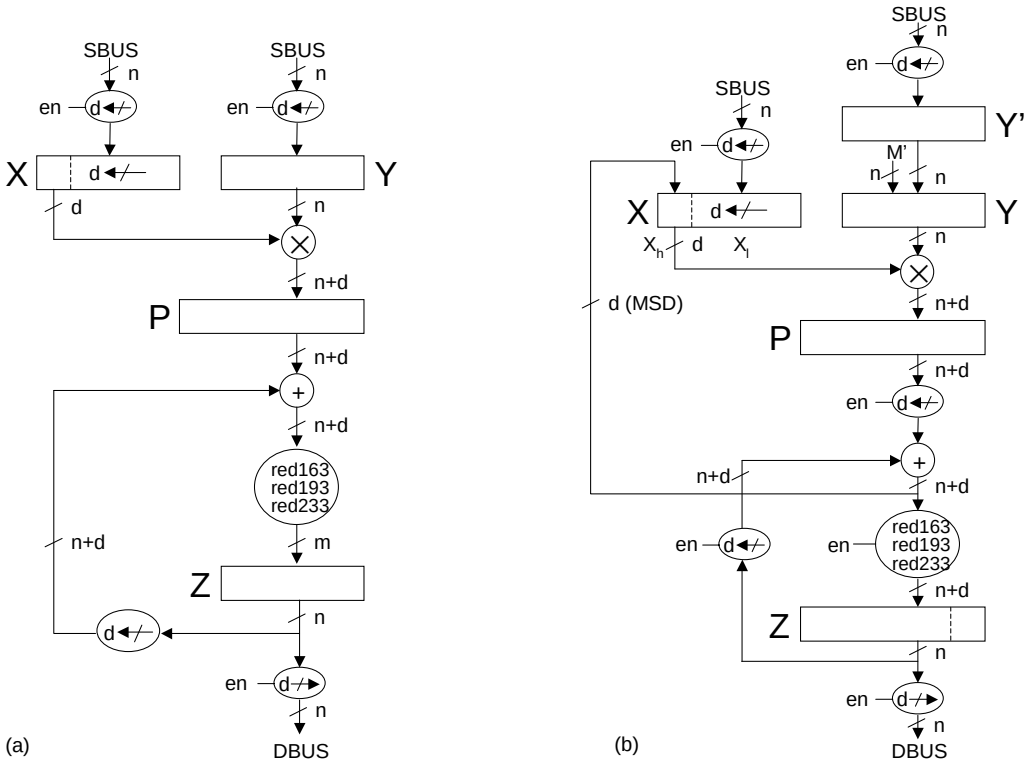


**Figure 3.** *MSD First Multiplier for Named Curves (a) and Generic MSD First Multiplier for Generic and Named Curves (b).*

We further developed a generic MSD first multiplier shown in Figure 3b that can handle both named and generic curves. It uses a hardwired reducer for named curves and it reuses the multiplier circuit to perform reduction for generic curves. Reduction for generic curves is based on the partial reduction algorithm. This technique reduces to the data path width $n$ rather than to the field size $m$, with $m \leq n$. This avoids costly shift and mask operations to extract field-sized operands smaller than the data path width. In comparison with Montgomery modular multiplication, our scheme uses fewer multiplications; this is particularly true when a large multiplier is used.

Partial reduction works as follow. For a multiplication $c_0 = a * b$ with $a, b \in GF(2^m)$, $\deg(a) < n$, $\deg(b) < n$, $c_0$ can be partially reduced to $c \equiv c_0 \mod M$, $\deg(c) < n$ as follows: For an integer $n \geq m$, $c_0$ can be split up into two polynomials $c_{0,h}$ and $c_{0,l}$ with $\deg(c_{0,h}) < n - 1$, $\deg(c_{0,l}) < n$. Subsequent polynomials $c_{j+1}$ can be computed by setting

$$c_{j+1} = c_{j,h} * t^{n-m} * (M - t^m) + c_{j,l} = c_{j+1,h} * t^n + c_{j+1,l}$$
$$\text{until} \quad c_{j,h} = 0 \Leftrightarrow \deg(c_j) < n$$

A more detailed description of partial reduction can be found in [9].

The pseudo code for performing modular multiplication on generic curves looks as follows:

```
X[n-1..0] := x*t^{d*⌊(n−m)/d⌋}; Y[n-1..0] := y*t^{d*⌊(n−m)/d⌋};
P[n+d-1..0] := 0; Z[n-1..0] := 0;
for i := 0 to ⌈m/d⌉-1 do
  P[n+d-1..0] := X[n-1..n-d] * Y[n-1..0];
  X[n-1..0] := shift_left(X[n-1..0],d);
  r[n+d-1..0] := shift_left(Z[n-1..0]) + P[n+d-1..0];
  Z[n-1..0] := r[n-1..0] + r[n+d-1..n] * (M − t^m) * t^{n−m};
end;
```

There is one partial product generator $\bigotimes$ that is alternately used to perform a multiplication step and a reduction step. Rather than strictly interleaving these two steps, the computation begins with executing two multiplication steps before the first reduction step is executed. That is, $P$ and $Z$ are computed in the order $\{P_0, P_1, Z_0, P_2, Z_1, ..\}$ such that $P_i$ is only needed two cycles later when $Z_{i+1}$ is calculated.

We also implemented a similar multiplier using the LSD first method. Comparing the two implementations we found that the MSD multiplier can be pipelined more efficiently saving one state for generic curves. The reason is that dependencies between partial products and reduction results are less stringent for the MSD first multiplier.

Note that we have assumed that it takes a single multiplication to execute a reduction step, which poses some restrictions on the irreducible polynomial. When reducing $r[n+d-1..0] = shift_left(Z[n-1..0]) + P[n+d-1..0]$, we assume that the partially reduced result of the multiplication $r[n-1..0] + r[n+d-1..n] * ((M - t^m) * t^{n-m})$ can be stored in an $n$-bit register. This requirement is equivalent to the partial reduction being executable in a single iteration. As explained in [9] this is true if $d \leq m - k$, where $k$ stands for the power of the second highest term of the irreducible polynoimal $M$. All polynomials recommended by NIST and SECG satisfy this condition.

### 4.1 Implementation

We prototyped the cryptographic processor in a Xilinx Virtex-II XCV2000E-7 FPGA. Area constraints were provided for the ALU, the divider and the register file, whereas the multiplier was left unconstrained. This way, these blocks do not interfere with each other when resources are allocated while, at the same time, as many resources as needed can be allocated to the multiplier which constitutes the critical path. No other constraints and, in particular, no manual placement was required to obtain a synthesized design that runs at the targeted frequency of 66.4 MHz which is derived from the PCI clock.

| Unit | LUTs | FFs |
|---|---|---|
| Generic MSD first Multiplier | 14797 | 2948 |
| ALU | 1345 | 279 |
| Divider | 2678 | 1316 |
| Full Design | 20068 | 6321 |

**Table 2.** *Usage of Chip Resources.*

Table 2 quantifies the resources used by the function units. The listed resources are 4-input look-up tables (LUTs) and flip-flops (FFs). The multiplier clearly dominates the size of the design as it uses 74% of the LUTs and 47% of the FFs. Since multiplication is the single most time-critical operation, its dominance, nevertheless, seems justified.

## 5 Point Multiplication Code

Various algorithms have been proposed to efficiently compute point multiplications [10]. We experimented with different point multiplication algorithms and settled on Mont-

gomery's point multiplication algorithm using projective coordinates as proposed by López and Dahab [11]. A fragment of the assembly code implementing the inner loop of Montgomery's double and add algorithm is shown in Table 3. The comments refer to the coordinates and curve parameters as found in [11]. The code is highly optimized in that the computation of the point coordinates is interleaved to achieve a higher degree of instruction-level parallelism. We use a single code base for named curves and generic curves. This is accomplished by executing MUL and SQR instructions according to the curve type. For named curves, MUL denotes a multiplication with hardwired reduction and, for generic curves, it is executed as a multiplication with partial reduction. The execution of an SQR instruction is slightly more complicated. For named curves, SQR is executed by the ALU. And for generic curves, the SQR instruction is translated into a MUL instruction that is executed as a multiplication using partial reduction. We use the BNC (branch if named curve) instruction in the few places where the program code differs for the two curve types.

As we had explained in Section 3.3 we make use of the fact that the multiplier and the ALU can operate in parallel. That is, if there are no data dependencies, the MUL instruction can be executed in parallel with either an ADD or a SQR instruction. Since the SQR instruction is executed by the ALU for named curves and by the multiplier for generic curves, the order in which instructions are executed differs depending on the curve type even though the code is the same.

Data dependencies are detected in different ways. The assembler checks for dependencies that would prevent overlapped instruction execution. In these cases, the programmer needs to resolve the dependencies by reordering operands or inserting NOP instructions. With respect to parallel instruction execution, the control unit examines dependencies and decides whether instructions can be executed in parallel or not.

| Instructions | | Execution for Named Curves | Execution for Generic Curves |
|---|---|---|---|
| | $R0 = X1, R1 = Z_1, R2 = X_2, R3 = Z_2$ | | |
| `MUL(R1,R2,R2)` | $R2 = Z_1 * X_2$ | `MUL(R1,R2,R2);SQR(R1,R1)` | `MUL(R1,R2,R2)` |
| `SQR(R1,R1)` | $R1 = Z_1^2$ | | `SQR(R1,R1)` |
| `MUL(R0,R3,R4)` | $R4 = X_1 * Z_2$ | `MUL(R0,R3,R4);SQR(R0,R0)` | `MUL(R0,R3,R4)` |
| `SQR(R0,R0)` | $R0 = X_1^2$ | | `SQR(R0,R0);ADD(R2,R4,R3)` |
| `ADD(R2,R4,R3)` | $R3 = Z_1 * X_2 + X_1 * Z_2$ | `ADD(R2,R4,R3)` | |
| `MUL(R2,R4,R2)` | $R2 = Z_1 * X_2 * X_1 * Z_2$ | `MUL(R2,R4,R2);SQR(R1,R4)` | `MUL(R2,R4,R2)` |
| `SQR(R1,R4)` | $R4 = Z_1^4$ | | `SQR(R1,R4)` |
| `MUL(R0,R1,R1)` | $R1 = Z_1^2 * X_1^2$ | `MUL(R0,R1,R1);SQR(R3,R3)` | `MUL(R0,R1,R1)` |
| `SQR(R3,R3)` | $R3 = Z_3 = (Z_1 * X_2 + X_1 * Z_2)^2$ | | `SQR(R3,R3)` |
| `LD(dmem_b,R5)` | $R5 = b$ | `LD(dmem_b,R5)` | `LD(dmem_b,R5)` |
| `MUL(R4,R5,R4)` | $R4 = b * Z_1^4$ | `MUL(R4,R5,R4);SQR(R0,R0)` | `MUL(R4,R5,R4)` |
| `SQR(R0,R0)` | $R0 = X_1^4$ | | `SQR(R0,R0)` |
| `LD(dmem_Px,R5)` | $R5 = X$ | `LD(dmem_Px,R5)` | `LD(dmem_Px,R5)` |
| `MUL(R3,R5,R5)` | $R4 = X * (Z_1 * X_2 + X_1 * Z_2)^2$ | `MUL(R3,R5,R5);ADD(R4,R0,R0)` | `MUL(R3,R5,R5);ADD(R4,R0,R0)` |
| `ADD(R4,R0,R0)` | $R0 = X_1^4 + b * Z_1^4$ | | |
| `ADD(R2,R5,R2)` | $R2 = X * Z_3 + (Z_1 * X_2) * (X_1 * Z_2)$ | `ADD(R2,R5,R2)` | `ADD(R2,R5,R2)` |

**Table 3.** *Code Execution for Named and Generic Curves.*

The code fragment in Table 3 shows no data dependencies for any MUL/SQR or MUL/ADD instruction sequence. Hence, for named curves, all MUL/SQR and MUL/ADD sequences are executed in parallel.

Furthermore, since there are no data dependencies between subsequent arithmetic instructions, instruction execution can be overlapped, thus, saving one cycle per instruction.

Code execution looks different for generic curves as illustrated. In this case, all MUL/SQR sequences have to be executed sequentially as SQR instructions are now executed as MUL

instructions. However, there still is one SQR/ADD sequence and one MUL/ADD sequence left that can be executed in parallel.

# 6 Evaluation

This section contains two parts. First, we look at the distribution of instructions executed by a point multiplication. Next, we compare performance numbers for point multiplication executed in hardware and software.

## 6.1 Instruction Distribution

Table 4 gives the distribution of instructions executed by the point multiplication operation for named and generic curves, respectively. For point multiplication on named curves over $GF(2^{163})$, field multiplications account for almost 62% of the execution time. In the case of generic curves, field multiplications even constitute 81% of the execution time. It is, therefore, justified to allocate a significant portion of the available hardware resources to the multiplier. Parallel and overlapped execution save 36% of the execution time for named curves and 20% for generic curves when compared to sequential execution. There is still room for further improvements since the control flow instructions BMZ, BEQ, SL, JMP and END consume almost 21% of the execution time when processing named curves and 10% when processing generic curves. This time could be saved by separating control flow and data flow.

| | Named Curves | | | Generic Curves | | |
|---|---|---|---|---|---|---|
| **Instruction** | **#Instr.** | **Cycles** | **ms** | **#Instr.** | **Cycles** | **ms** |
| MUL | 6 | 41 | 0.00062 | 818 | 7366 | 0.11093 |
| MUL + ADD | 166 | 996 | 0.01500 | 327 | 2943 | 0.04432 |
| MUL + SQR | 811 | 4866 | 0.07328 | 0 | 0 | 0.00000 |
| SQR | 2 | 4 | 0.00006 | 651 | 5859 | 0.08824 |
| DIV | 1 | 329 | 0.00495 | 2 | 902 | 0.01358 |
| ADD | 330 | 660 | 0.00994 | 170 | 340 | 0.00512 |
| SL | 326 | 978 | 0.01473 | 326 | 978 | 0.01473 |
| ST | 6 | 18 | 0.00027 | 8 | 24 | 0.00036 |
| LD | 334 | 668 | 0.01006 | 337 | 674 | 0.01015 |
| BMZ | 326 | 652 | 0.00982 | 326 | 652 | 0.00982 |
| BEQ | 1 | 4 | 0.00006 | 1 | 4 | 0.00006 |
| BNC | 2 | 4 | 0.00006 | 2 | 4 | 0.00006 |
| JMP | 162 | 324 | 0.00488 | 162 | 324 | 0.00488 |
| END | 1 | 2 | 0.00003 | 1 | 2 | 0.00003 |
| total | | 9549 | 0.14381 | | 20072 | 0.30229 |

**Table 4.** *Decomposition of the Execution Time for $GF(2^{163})$ Point Multiplication.*

## 6.2 Point Multiplication Performance

Table 5 shows performance numbers for implementations of point multiplication in hardware and software. The hardware implementation uses the prototype system described in Section 4.1. The software implementation considers generic curves and does not contain any curve-specific optimizations. It is executed on a 900 MHz Sun Fire™280R server. The execution time of a point multiplication $kP$ depends on the execution times of the arithmetic operations in $GF(2^m)$ and the size of the integer $k$, which is mostly in the order of

the field degree $m$. The time needed for a point multiplication grows almost linearly with the size of $k$ between $1 \leq m \leq 192$ and $193 \leq m \leq 256$. The non-linear increase at $m = 192$ is caused by the multiplier that exhibits different execution times based on the field degree.

By adding hardware support for generic curves, we were able to significantly increase performance for generic curves. In [9] we described a firmware implementation with a performance number of 1075 point multiplications per second for generic curves over $GF(2^{163})$. At 3308 point multiplications per second, our new design is roughly three times faster. Compared with 6955 point multiplications per second for named curves over $GF(2^{163})$, the performance penalty for named curves is now roughly a factor of two which is low given the complexity of the problem.

| Curves | Hardware | | Software | | Speedup |
|---|---|---|---|---|---|
| | ops/s | ms/op | ops/s | ms/op | |
| **Named** | | | | | |
| $GF(2^{163})$ | 6955 | 0.14 | 322 | 3.11 | 21.6 |
| $GF(2^{193})$ | 5333 | 0.19 | 294 | 3.40 | 18.1 |
| $GF(2^{233})$ | 4423 | 0.23 | 223 | 4.48 | 19.8 |
| **Generic** | | | | | |
| $GF(2^{163})$ | 3308 | 0.30 | | | |
| $GF(2^{193})$ | 2375 | 0.42 | | | |
| $GF(2^{233})$ | 1980 | 0.51 | | | |

**Table 5.** *Hardware and Software Performance.*

The speedup of the hardware implementation over the software implementation is a factor of about 20 for named curves. The poor software performance is mainly due to the lack of support for $GF(2^m)$ arithmetic in general-purpose CPUs. While the software implementation is optimized for irreducible polynomials that are either pentanomials or trinomials, the hardware implementation is more generic in that it can operate on arbitrary irreducible polynomials.

## 7 Conclusions

We presented an ECC processor that provides optimized performance for a number of named curves and support for generic curves over arbitrary fields $GF(2^m), m \leq 255$. This flexibility is needed by server applications that have to perform large numbers of point multiplications on different curves.

We described a novel modular multiplier that is capable of handling named curves as well as generic curves. Processing the two types of curves differs in that hardwired reduction logic is used for named curves and the multiplier logic is reused to perform reduction for generic curves. Since reduction for generic curves reuses existing logic, the additional resources needed to support generic curves are minimal. We implemented a generic MSD first multiplier with operand size $n = 256$ and digit size $d = 64$. Including the cycles needed for loading and storing operands, it takes seven cycles to perform a modular multiplication for named curves over fields $GF(2^m), m \leq 192$, eight cycles for named curves over fields $GF(2^m), 193 \leq m \leq 255$, 10 cycles for generic curves over fields $GF(2^m), m \leq 192$, and 12 cycles for generic curves over fields $GF(2^m), 193 \leq m \leq 255$.

Our ECC processor uses a common code base to implement point multiplication for both named curves and generic curves. To make this possible, squaring instructions are

dynamically translated into multiplication instructions in the case of generic curves since modular squaring is implemented in hardware for named curves only.

We optimized performance by exploiting parallelism found in Montgomery's point multiplication algorithm for projective coordinates. More specifically, we allow multiplication instructions, which are the most frequently executed instructions, to be executed in parallel with either add or square instructions. Together with overlapped instruction execution, parallel execution reduces the execution time for a point multiplication operation by 36% for named curves and 20% for generic curves.

We are currently working on a cryptographic processor that exploits a common architecture that is capable of executing point multiplication for both $GF(p)$ and $GF(2^m)$.

## References


[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^{155}}$. *In IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[2] M. Bednara, M. Daldrup, J. von zur Gathen, and J. Shokrollahi. Reconfigurable implementation of elliptic curve crypto algorithms. *Reconfigurable Architectures Workshop, 16th International Parallel and Distributed Processing Symposium*, April 2002.

[3] Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Standards for Efficient Cryptography Version 1.0, September 2000.

[4] S. Chang-Shantz. From euclid's gcd to montgomery multiplication to the great divide. Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001. http://research.sun.com/.

[5] J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[6] V. Gupta, S. Blake-Wilson, B. Möller, and C. Hawk. *ECC Cipher Suites for TLS.* IETF Internet Draft, August 2002. http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-02.txt.

[7] V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for ssl. In *ACM Workshop on Wireless Security*, September 2002. Atlanta, Georgia.

[8] N. Gura, H. Eberle, and S. Chang-Shantz. An end-to-end systems approach to elliptic curve cryptography. In *CHES '2002 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science. Springer-Verlag, August 2002. Redwood City, California.

[9] N. Gura, H. Eberle, and S. Chang-Shantz. Generic implementations of elliptic curve cryptography using partial reduction. In *9th ACM Conference on Computers and Communications Security*, November 2002. Washington, DC.

[10] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[11] J. López and R. Dahab. Fast multiplication on elliptic curves over $\mathbb{GF}(2^m)$ without precomputation. In *CHES '99 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1717. Springer-Verlag, August 1999.

[12] OpenSSL Project. http://www.openssl.org/.

[13] G. Orlando and C. Paar. A high-performance reconfigurable elliptic curve processor for $\mathbb{GF}(2^m)$. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[14] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *IEEE Journal of VLSI Signal Processing Systems*, (19):149–166, 1998.

[15] U.S. Department of Commerce and National Institute of Standards and Technology. Digital signature standard (dss). Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[16] A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *The Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, September 2000. Bristol, UK.