Implementation of Elliptic Curve Cryptosystems on a Reconfigurable Computer

Nghi Nguyen[1], Kris Gaj[1], David Caliga[2], Tarek El-Ghazawi[3]
[1] George Mason University, [2] SRC Computers, [3] The George Washington University

Abstract. During the last few years, a considerable effort has been devoted to the development of reconfigurable computers, machines that are based on the close interoperation of traditional microprocessors and Field Programmable Gate Arrays (FPGAs). Several prototype machines of this type have been designed, and demonstrated significant speedups compared to conventional workstations for computationally intensive problems, such as codebreaking. Nevertheless, the efficient use and programming of such machines is still an unresolved problem. In this paper, we demonstrate an efficient implementation of an Elliptic Curve scalar multiplication over GF(2^m), using one of the leading reconfigurable computers available on the market, SRC-6E. We show how the hardware architecture and programming model of this reconfigurable computer has influenced the choice of the algorithm partitioning strategy for this application. A detailed analysis of the control, data transfer, and reconfiguration overheads is given in the paper, together with the performance comparison of our implementation against an optimized microprocessor implementation.

Keywords: Reconfigurable Computers, FPGA devices, Elliptic Curve Cryptosystems, Galois Fields

1. Introduction

Reconfigurable Computers are general-purpose high-end computers based on a hybrid architecture and close system-level integration of traditional microprocessors and Field Programmable Gate Arrays (FPGAs). It is desired that programming of reconfigurable computers should not require any knowledge of hardware design, assuming that a sufficiently large library of elementary operations has been earlier developed and made available to programmers.

The emergence of reconfigurable computers offers a great promise in terms of progress in many traditionally hard cryptographic problems. Many problems, such as integer factorization, elliptic curve discrete logarithm problem, or counting the number of points on an elliptic curve have been shown in theory to execute substantially more efficiently in hardware [1, 2]. At the same time, no prototypes confirming these claims have been reported in the open literature for practical sizes of cryptographic parameters because of the prohibitive cost of specialized hardware.

Although a lot of work has been done in the area of reconfigurable computing and run-time reconfiguration, we are aware of only few practical implementations of general-purpose reconfigurable computers. SRC-6E from SRC Computers, Inc. was chosen for our study [3]. Our goal was not only to confirm the great potential for effective use of reconfigurable computers in cryptography, but also to determine the current and possible future limitations of the reconfigurable computing technology. We chose as our benchmark a relatively complex cryptographic operation: scalar multiplication in the group of points on an elliptic curve over GF(2^m) with a polynomial basis representation [4, 5, 6]. This operation is perfect for our study, as it involves a three-level hierarchy of operations. The goal of our study is to find out which level functions need to be implemented by a hardware designer as library macros, and at what level the software designer can take over. Our paper gives an answer to this question for the current generation of reconfigurable computers.

2. SRC Reconfigurable Computer

SRC-6E is a hybrid-architecture platform, which consists of two dual-microprocessor boards and one MAP® board. A block diagram depicting a half of the SRC-6E machine is shown in Fig. 1. Each microprocessor board is connected to the MAP board through the SNAP® interconnect, which can support a peak bandwidth of 800 MB/s.

SRC-6E has a similar compilation process to a conventional microprocessor-based computing system, but needs to support additional tasks in order to produce logic for the MAP reconfigurable processor, as shown in Fig. 2. There are two types of the application source files to be compiled. Source files of the first type are compiled targeting execution on the Intel microprocessors. Source files of the second type are compiled targeting execution on the MAP processor. These MAP source files contain functions composed of HLL instructions and HDL macro calls. Such functions will be referred to in this article as MAP functions. Here, macro is defined as a piece of hardware logic designed to implement a certain function. Since users often wish to extend the built-in set of operators, the compiler allows users to integrate their own macros, encoded in VHDL

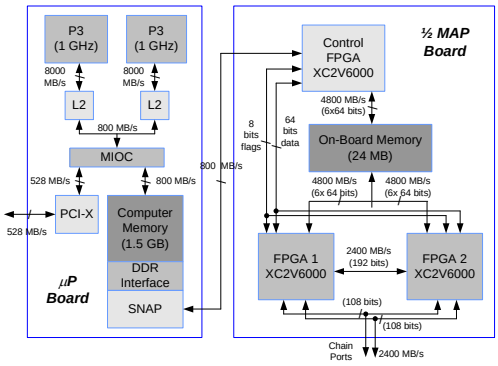

Fig. 1. Hardware architecture of SRC-6E
Notation:
P3 – Intel Pentium 3 Microprocessor,
L2 – Level 2 Cache,
MIOC – Memory and I/O Bridge Controller
PCI – Peripheral Component Interconnect Interface
DDR Interface – Double Data Rate Memory Interface
SNAP – SRC-developed Memory Interconnect
MAP - Reconfigurable Processor

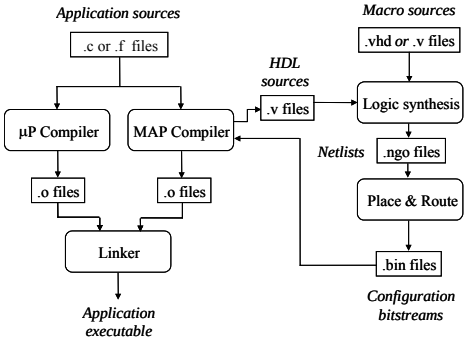

Fig. 2. Compilation process of SRC-6E

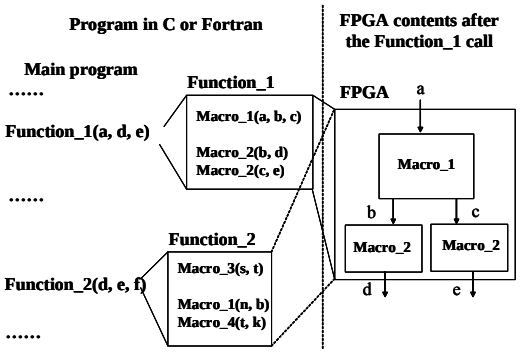

Fig. 3. Programming model of SRC-6E

or Verilog, into the compilation process. All macros must be optimized to operate at the clock frequency of 100 MHz. A macro is invoked from within the C or Fortran function by means of a function call.

In Fig. 3, we demonstrate the mapping between macro calls and the corresponding contents of a MAP FPGA. Please, note that Macro 2, called twice in Function 1, results in two instantiations of the logic block representing Macro 2. Values of arguments in the macro calls determine interconnects between macro instantiations in hardware. The contents of each MAP function in software determines the configuration of the entire FPGA device in hardware. Each time a new MAP function is called, the contents of the entire FPGA changes. If the same function is called multiple number of times in sequence, the reconfiguration is performed only once. During the subsequent function calls, only data and control transfers take place. This way, SRC-6E implements run-time reconfiguration.

An application can be implemented either using a single User FPGA, or partitioned among two User FPGAs available on the MAP board. The communication between these two FPGAs is performed using a 192-bit bridge port and auxiliary communication macros.

3. Basic operations of Elliptic Curve Cryptosystems

Elliptic Curve Cryptosystems (ECCs) are used commonly in constrained environments, such as portable and wireless devices, as a small-area, low-energy alternative to the RSA cryptosystem. The primary application of Elliptic Curve Cryptosystems is secure key agreement and digital signature generation and verification [5, 7, 8]. In both of these applications the primary optimization criterion from the implementation point of view is the minimum latency (rather then the maximum throughput). The primary operation of ECCs is an *elliptic curve scalar multiplication*. Below we define this operation in terms of lower level operations.

A non-supersingular elliptic curve over GF(2^m) is defined as set of points (x,y) that satisfy the equation,

$$y^2 + xy = x^3 + a_2x^2 + a_6 , \quad (1)$$

where, x, y, $a_6 \in$ GF(2^m), and $a_2 \in \{0,1\}$, together with the special point called a point at infinity, and denoted as *O*.

The elements of the Galois Field GF(2^m) can be represented in several different bases, such as polynomial basis, normal basis, dual basis, etc. In all these representations, addition is the same and equivalent to the XOR operation, but multiplication is defined differently. Our implementation focuses on the polynomial basis representation.

An addition of two points of an elliptic curve P=(x_P, y_P) and Q=(x_Q, y_Q), where Q≠-P=(x_P, y_P+x_P), and P, Q ≠ O is defined in Table 1. Additionally, P+ O = O + P = P, and P + (-P) = O. Similarly, point doubling, 2P=P+P, where P≠ O, is also defined in Table 1. Additionally, 2 O = O. Please, note that outside of special cases, both point addition and point doubling involve one inversion, several multiplications, and several additions in GF(2^m).

Table 1. Formulas for the point addition and doubling for elliptic curves over GF(2^m)

Point Addition	Point Doubling
$x_R = \lambda^2 + \lambda + x_P + x_Q + a_2$ $y_R = \lambda \cdot (x_P + x_R) + x_R + y_P$ $\lambda = (y_Q + y_P) \cdot (x_Q + x_P)^{-1}$ # Inv: 1 # Mul: 3 # Add: 8	$x_R = a_6 (x_P^{-1})^2 + x_P^2$ $y_R = x_P^2 + (x_P + y_P \cdot x_P^{-1}) \cdot x_R + x_R$ # Inv: 1 # Mul: 5 # Add: 4

The primary elliptic curve operation used in cryptography is scalar multiplication, defined as

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

A very well known right-to-left and left-to-right algorithms for scalar multiplication are given below as Algorithms 1 and 2. In Algorithm 1, point addition (line 5) and point doubling (line 7) can be performed in parallel. The same is not true for Algorithm 2. Therefore, we have chosen the right-to-left Algorithm 1 for our implementations.

Algorithm 1 Right-to-left scalar multiplication

Input: P = (x_P, y_P),
k = (k_{m-1}, k_{m-2}, ..., k_1, k_0)$_2$,
where $0 \le k < q$ q = order of point P
Output: R = kP
Auxiliary: S = (x_S, y_S)

```
1: R = O
2: S = P
3: for ( i=0 to m-1 )
4:     if( k_i = 1 )
5:         R = R + S
6:     end if
7:     S = 2S
8: end for
9: return R
```

Algorithm 2 Left-to-right scalar multiplication

Input: P = (x_P, y_P),
k = (k_{m-1}, k_{m-2}, ..., k_1, k_0)$_2$,
where $0 \le k < q$ q = order of point P
Output: R = kP

```
1: R = O
2: for ( i=m-1 downto 0 )
3:     R = 2R;
4:     if ( k_i = 1 ) then
5:         R = R+ P
6:     end if
7: end for
8: return R
```

4. Investigated Partitioning Schemes

A hierarchy of operations involved in an elliptic curve scalar multiplication for the case of an elliptic curve over GF(2^m) is given in Fig. 4. Three levels of operations are involved in this hierarchy: scalar multiplication, kP, at the high level (H), point addition and point doubling at the medium level (M), and the GF(2^m) multiplication (MUL), inversion (INV), and addition (XOR) at the low level (L).

Functions belonging to each of these three hierarchy levels (high, medium, and low) can be implemented using three different implementation approaches:

a) as a C function compiled for a general-purpose microprocessor,
b) as a C function compiled by the SRC MAP compiler to the hardware description code running on the User FPGA, and
c) as a VHDL hardware macro running on the User FPGA.

Two possible extreme cases are to implement scalar multiplication kP entirely in software as a C microprocessor function, or entirely in hardware, using traditional hardware design methodology (i.e., as a VHDL hardware macro, as shown in Fig. 5d). Several intermediate partitioning schemes are possible, and are presented schematically in Figs. 5abc.

Each of these approaches is characterized by a three letter codename, such as HML, 0HL, 0HM, etc. The first letter of this codename determines which level operations (high, medium, low, or none) are implemented in C on a general-purpose microprocessor. The second letter, determines which operations are described as a C function for the MAP, and the third

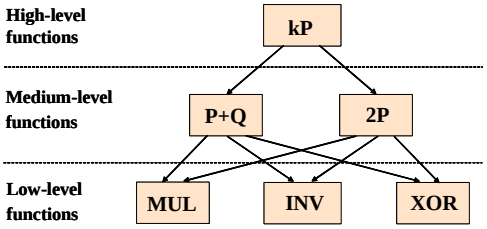

Fig. 4. Hierarchy of the ECC operations

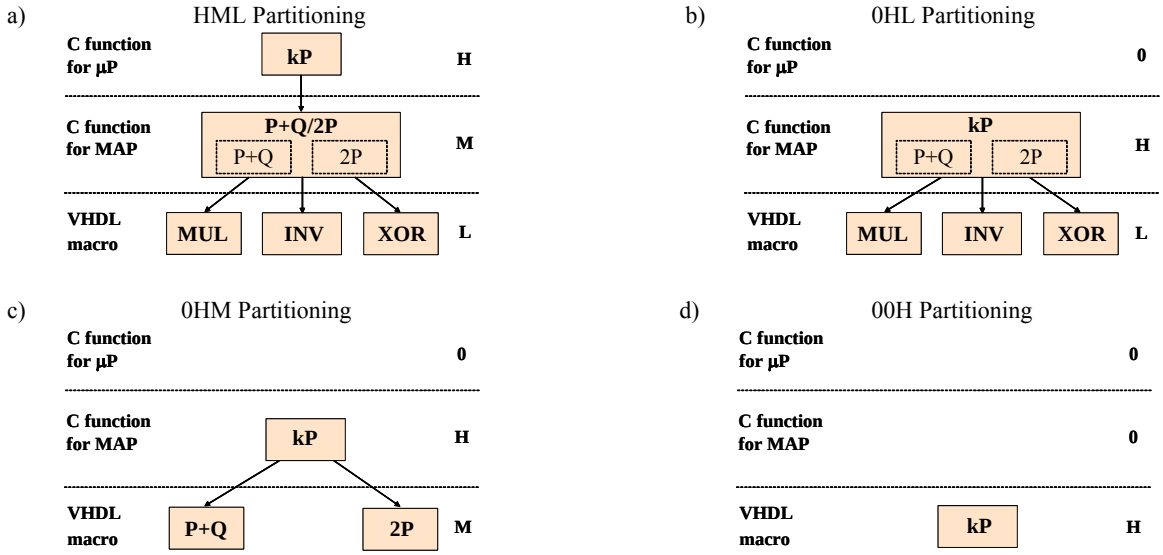

Fig. 5. Four alternative algorithm partitioning schemes

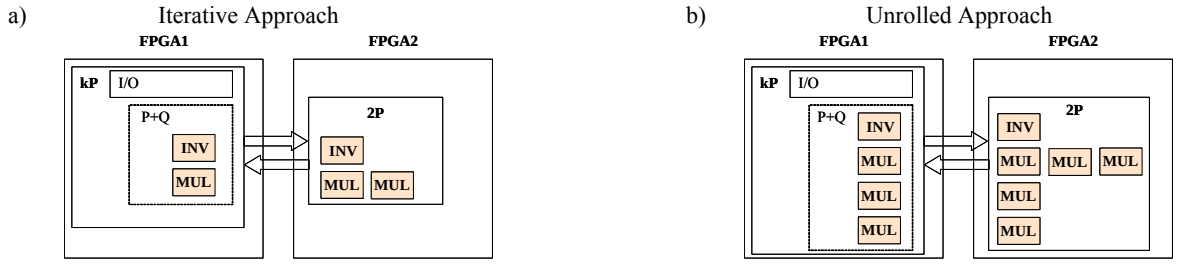

Fig. 6. FPGA design partitioning for two alternative implementation approaches of the 0HL scheme

letter, which operations are implemented as HDL macros. For example, the codename 0HL means that no operations are implemented in C for the microprocessor, a high-level operation (kP) is implemented as a C function for the MAP, and low level operations (MUL, INV, XOR) are implemented as VHDL macros.

The first, straightforward partitioning approach, HML, is shown in Fig. 5a. In this approach, the C MAP function performs in parallel two medium-level operations, P+Q and 2P. The results of both of these operations are returned to kP. Here, based on a value of the currently processed bit of k, k_i, the result of the point addition is either discarded, or used to update the intermediate result R in Algorithm 1 (see Algorithm 1, lines 4-6). The result of doubling is always used to update the auxiliary variable S (see Algorithm 1, line 7). Please, note that based on the SRC programming model (explained in Section 2), if P+Q and 2P were implemented as separate MAP functions, then the reconfiguration of the User FPGA would need to take place each time we switch execution between P+Q and 2P. Since the time of the reconfiguration of the User FPGAs has been measured to be equal to about 97 ms, and kP implemented in VHDL executes within only 3 ms, even a

single reconfiguration time by far exceeds the total execution time of kP in hardware. The existence of an integrated P+Q/2P function and calling this function once as a part of the application setup eliminates the reconfiguration overhead.

Unfortunately, an additional timing overhead is introduced during each MAP function call because of the control, input, and output transfer between the microprocessor board and the MAP board. In the current generation of the SRC system, this overhead has been measured to be in the range of 370 μs. This value is very large compared to the average execution time of the P+Q and 2P operations in hardware (in the range of 10-20 μs).

In order to minimize this overhead, the 0HL partitioning scheme (shown in Fig. 5b) has been implemented. In this scheme, the MAP function is called only once and executes the entire high level operation kP. As a result, the control, input, and output overheads are decreased, on average, by a factor of m, i.e., by at least two orders of magnitude for practical values of m (such as m=233 used in our experiments).

Two possible implementation approaches have been considered in the case of the 0HL partitioning: the iterative and the unrolled. In the iterative approach (see Fig. 6a), only one multiplier instantiation is used to implement the P+Q operation, and two multiplier instantiations are used to implement the 2P operation. These multipliers are used iteratively to perform a total of 3 multiplications per P+Q operation, and 5 multiplications per 2P operation. In the unrolled approach (see Fig. 6b), the number of instantiations of the multiplication macro is the same as the number of multiplications to be performed. The iterative approach is more efficient in terms of the circuit area, and exploits the fact that only a limited number of multiplications can be executed in parallel because of the data dependencies between subsequent multiplications. On the other hand, the unrolled approach simplifies control logic and reduces circuit latency. From the programming point of view, both approaches require a similar amount of effort.

A further reduction in the execution time can be accomplished in the 0HM partitioning shown in Fig. 5c, by implementing medium level operations, P+Q and 2P, as VHDL macros. The disadvantage of this approach is the required hardware knowledge, the level of HDL programming experience and the increased effort necessary to develop VHDL code in place of the C function for the MAP. The advantage is the opportunity for manual optimization of the VHDL code versus the HDL generated by the SRC MAP compiler. This process is analogous to doing assembly coding for the microprocessor. This hardware-oriented approach can be taken to its extreme by implementing the entire kP operation as VHDL macro (see partitioning scheme 00H shown in Fig. 5d).

Each of the aforementioned partitioning schemes can be implemented in principle using either a single User FPGA or two User FPGAs. In case two FPGA devices are used, the first one is used to implement P+Q, input/output, and possibly control operations for kP (for the 0HL and 0HM approaches), and the second one is used to implement 2P, as shown in Fig. 6 for the two 0HL implementation approaches.

5. Implementation of Multiplication and Inversion in GF(2^m)

Multiplication in GF(2^m) with polynomial basis representation is defined as follows. Inputs $A = (a_0, a_1, \ldots, a_{m-1})$ and $B = (b_0, b_1, \ldots, b_{m-1}) \in$ GF(2^m), and the product $C = AB = (c_0, c_1, \ldots, c_{m-1})$ are treated as polynomials A(x), B(x), and C(x) with respective coefficients. The dependence between these polynomials is given by

$$C(x) = A(x) \cdot B(x) \bmod P(x),$$

where P(x) is a constant irreducible polynomial of degree m. The straightforward shift-and-add algorithm for multiplication in GF(2^m) is given below as Algorithm 3, and its implementation is presented in Fig. 7. This algorithm has been selected because it easily supports 100 MHz clock frequency required by the SRC system. In our implementation, this multiplier performs a single multiplication in m+1 clock cycles.

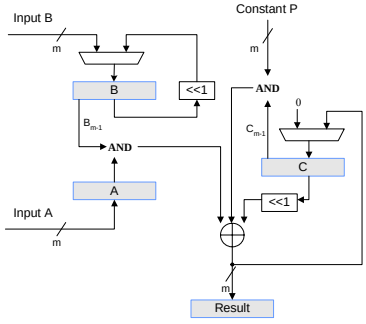

Fig. 7. Multiplier in GF(2^m)

Algorithm 3 Multiplication in GF(2^m) with interleaved modular reduction

Input: A(x), B(x) ∈ GF(2^m)
P(x) irreducible polynomial of degree m
Output: C(x) = A(x) * B(x) mod P(x)

1: C(x) ⇐ 0
2: **for** (i=m-1 downto 0)
3: C(x) ⇐ C(x)*x + A(x)b_i
4: C(x) ⇐ C(x) + c_mP(x)
5: **end for**
6: **return** C(x)

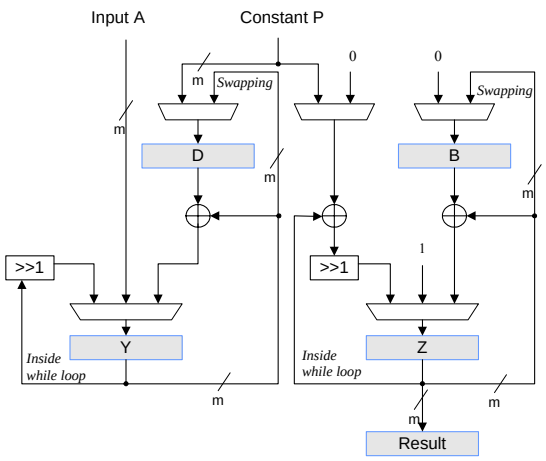

Fig. 8. Inverter in GF(2^m) based on the Modified Almost Inverse Algorithm

Algorithm 4 Inversion in GF(2^m)
Modified Almost Inverse Algorithm

Input: $A(x) \in GF(2^m)$, $A(x) \neq 0$
$P(x)$ irreducible polynomial of degree m
Output: $C(x) = A(x)^{-1} \bmod P(x)$

1: $Y(x) \Leftarrow A(x)$, $D(x) \Leftarrow P(x)$, $B(x) \Leftarrow 0$, $X(x) \Leftarrow 1$
2: **loop**
3: **while** ($y_0 = 0$)
4: $Y(x) \Leftarrow Y(x) * x^{-1}$
5: $X(x) \Leftarrow (X(x) + x_0 P(x)) * x^{-1}$
6: **end while**
7: **if** ($Y(x) = 1$) **then**
8: **return** $X(x)$
9: **end if**
10: **if** (deg $Y(x)$ < deg $D(x)$) **then**
11: $Y(x) \Leftrightarrow D(x)$, $B(x) \Leftrightarrow X(x)$
12: **end if**
13: $Y(x) \Leftarrow Y(x) + D(x)$, $X(x) \Leftarrow X(x) + B(x)$
14: **end loop**

Our implementation of inversion in GF(2^m) is based on the Modified Almost Inverse Algorithm presented in [9, 10]. This algorithm is given above as Algorithm 4, and its implementation is presented in Fig. 7. This inverter has a variable, data-dependent latency in the range of 3·m clock cycles.

6. Design methodology and testing

Our implementations are capable of handling elliptic curve operations over GF(2^m) for m ≤ 256. In particular, m=233 was chosen for our experiments, as this is one of the sizes recommended by NIST [7]. Additionally, our implementation can be easily extended to process larger values of m by using multiple memory locations to store a single element of GF(2^m).

All hardware macros have been developed first using standard tools for simulation and synthesis of digital circuits, such as Aldec Active-HDL and Synplicity Synplify Pro. All macros have been optimized to work at the clock frequency of 100 MHz. The 'XOR' operation did not need to be implemented as a user macro, as it is a standard macro in the SRC library. This macro is invoked automatically when compiler encounters the XOR operator (denoted as '^') within a C MAP function.

All our implementations have been tested for correct functionality using an optimized software implementation based on LiDiA, the public domain library for computational number theory [11]. The execution time of operations within the C MAP function has been measured in the number of clock cycles using the standard SRC macro, read_timer(). The end-to-end time of C functions has been measured in time units using the C timer function of the Linux operating system, gettimeofday(). All measurements have been repeated 100 times, and the median values are reported in Section 7.

7. Results

The results of the timing measurements for all investigated partitioning schemes are summarized in Table 2. The FPGA Computation Time includes only the time spent performing computations using User FPGAs. The End-to-End time includes the FPGA Computation time and all overheads associated with the data and control transfers between the microprocessor board and the FPGA board. The Total Overhead is the difference between the End-to-End time and the FPGA Computation Time. Two specific components of the Total Overhead listed in Table 2 are DMA Data In Time, and DMA Data Out Time. They represent, respectively, the time spent to transfer inputs from the Common Memory to the On-board Memory, and the time spent to transfer outputs from the On-Board Memory to the Common Memory.

On two extremes, Table 2 shows the End-to-End Time for the purely software implementation (Architecture H00), equal to about 31 ms, and the FPGA Computation Time for the purely VHDL implementation (Architecture 00H), equal to about 3 ms. The speed-up by a factor of ten has been demonstrated. It should be noted, that this speed-up could be several times greater, if we considered throughput (number of scalar multiplications per unit of time), instead of latency, and used all resources available in User FPGAs for implementation of multiple computational units working in parallel. Additionally, our

Table 2. Results of the timing measurements for several investigated partitioning schemes and implementation approaches

Algorithm Partitioning Scheme	No. of FPGAs	End-to-End Time (µs)	DMA Data In Time (µs)	FPGA Computation Time (µs)	DMA Data Out Time (µs)	Total Overhead (µs)	Speed-up vs. Software	Slow-down vs. VHDL macro
H00 (software)	N/A	**31,050**	N/A	N/A	N/A	N/A	**1.00**	**9.27**
HML	2 chips	101,145	9,683	3,710	2,751	89,630	**0.31**	**30.2**
0HL iterative	2 chips	3,914	40	3,544	15	370	**7.93**	**1.17**
0HL unrolled	2 chips	3,615	40	3,247	11	368	**8.59**	**1.08**
0HM	1 chip	4,936	41	4,565	13	371	**6.29**	**1.47**
	2 chips	3,522	41	3,154	10	368	**8.82**	**1.05**
00H (VHDL)	1 chip	3,349	42	**2,979**	12	370	**9.27**	**1.00**

comparison assumes that the general purpose microprocessor is dedicated entirely to performing cryptographic transformations, which is rarely the case.

The most straightforward HML partitioning scheme is over three times slower than software, and over 30 times slower than VHDL macro. The execution time in this scheme is dominated by the Total Overhead that includes preparing and transferring a list of commands for the Control FPGA (the ComList), transferring inputs from the Common Memory to the On-board Memory, transferring outputs from the On-Board Memory to the Common Memory, and additional control operations. Since this overhead is about 24 times larger than the execution time of medium-level functions within MAP, it dominates the End-to-End Execution time. Based on this example, we have demonstrated the importance of taking overhead into account when determining partitioning boundaries. The performance impact can be dramatic. For the current generation of the SRC-6E system, implementing elliptic curve point addition, P+Q, and point doubling, 2P, as a MAP function (the HML partitioning) results in interface overhead that dominates the computation. As the microprocessor to MAP interface overhead is reduced, the HML partitioning may become a viable implementation.

The immediate solution is to implement the entire kP operation as a single MAP function, as in the partitioning scheme 0HL. If this scheme is implemented using two FPGA devices, and unrolled (see Fig. 6b), then its End-to-End Time is only 8% greater than the End-to-End time of the purely VHDL version. A much more difficult to implement 0HM scheme, which requires developing VHDL code for the P+Q and 2P operations, gives only marginal speed-up over the 0HL unrolled scheme, assuming that both implementations are partitioned over two User FPGAs. Finally, even if the entire code for kP is written in VHDL, the data and control transfer overheads still contribute to a 12% increase in the End-to-End Execution Time over the pure FPGA Computation Time.

The choice of the optimum architecture might be different if the use of the FPGA resources is of concern. This use in terms of the percentage of the CLB (Configurable Logic Block) slices, LUTs (Look-Up Tables), and FFs (Flip Flops) is summarized in Table 3. In particular, the unrolled 0HL scheme implemented using two chips takes 3.3 times more CLB slices, 1.6 times more LUTs, and 2.3 times more flip-flops than the purely VHDL version. Future versions of the compiler are targeted to dramatically reduce this CLB count. If the latency is of primary concern, and machine is used exclusively

Table 3. Resource utilization for several investigated partitioning schemes and implementation approaches

Algorithm Partitioning Scheme	No. of FPGAs	% of CLB slices (out of 33,792)	CLB count Increase vs. pure VHDL	% of LUTs (out of 67,580)	LUT count increase vs. pure VHDL	% of FFs (out of 67,580)	FF count increase vs. pure VHDL
H00 (software)	N/A	0	N/A	0	**N/A**	0	N/A
HML	2 chips	57+48 = 105	**2.69**	19+15 = 34	**1.55**	40+31= 71	**2.29**
0HL iterative	2 chips	59+48 = 107	**2.74**	19+16=35	**1.59**	40+31=71	**2.29**
0HL unrolled	2 chips	79+49 = 128	**3.28**	20+14= 34	**1.55**	44+28= 72	**2.32**
0HM	1 chip	46	**1.18**	23	**1.05**	36	**1.16**
	2 chips	37+14 = 51	**1.31**	17+9= 26	**1.18**	31+11= 42	**1.35**
00H (VHDL)	1 chip	39	**1.00**	22	**1.00**	31	**1.00**

for the ECC operations, this increase is inconsequential, as the remaining FPGA resources remain unused anyway. If however the multiple instantiations of the same architecture are to be implemented for increased throughput, or User FPGAs are expected to be used for other operations as well, then the 0HL iterative architecture, or even a 0HM architecture might be a better choice.

The 0HM scheme is more difficult to implement than 0HL unrolled scheme, because of the additional operations needed to be expressed in VHDL (approximately twice as many lines of VHDL code). Nevertheless, using this scheme gives substantial advantages in terms of resource usage (e.g., reduction in the number of CLB slices by a factor of 2.5 compared to the unrolled 0HL scheme), without imposing any time penalty.

The current version of the MAP compiler (SRC-6E Carte 1.4.1) optimizes performance over resource utilization. As it matures the compiler will be expected to balance high performance, ease of coding, and resource utilization to yield a truly optimized logic.

8. Conclusions

Reconfigurable computers offer a great promise for solving complex cryptographic problems with the speed of specialized hardware and flexibility and productivity of software implementations. In this paper, we describe our experiences with programming one of the leading reconfigurable computers available on the market, SRC-6E.

We have chosen as our benchmark the primary operation of Elliptic Curve Cryptosystems over GF(2^m) in polynomial basis representation: scalar multiplication. This operation is particularly challenging for reconfigurable computers because the primary optimization criterion is latency rather than throughput, and there is only limited amount of parallelism involved in the medium level operations, such as point addition and doubling. In spite of these constraints, a speed-up in the range of 8-9 has been demonstrated compared to the highly optimized microprocessor implementation using four different algorithm partitioning approaches (0HL iterative 2-chip, 0HL unrolled 2-chip, 0HM 2-chip, and 00H 1-chip).

What is more important however, our study revealed the optimum boundary between hardware and software, and between the descriptions of hardware in VHDL vs. C for the three-level hierarchy of operations constituting the Elliptic Curve scalar multiplication. This boundary had to take into account the trade-off between the end-to-end execution time, the resource utilization, and the designer's productivity and ability. While the first two criteria are relatively easy to quantify, the third one is more difficult to measure objectively, as it depends strongly on the designer's skills and background. Additionally, the relative importance and weight of particular criteria might very depending on particular application and design environment.

Assuming as a primary criterion the increased application developer productivity and an attempt to minimize involvement of hardware designers and traditional HDL-based design methodology, we have determined an optimum solution. In this solution, referred to as unrolled 0HL scheme, the entire scalar multiplication is implemented in hardware, but only low-level operations, GF(2^m) multiplication and inversion, needed to be described in VHDL. This partitioning scheme was shown to increase the execution time only by 8% compared to the scheme based on implementing the entire scalar multiplication in VHDL. This result was accomplished at the cost of the increased use of FPGA resources, such as CLB slices, used mostly as a source of additional flip-flops.

Our research demonstrated that a good knowledge of the system hardware architecture and programming model of a reconfigurable computer, and the associated overheads, may be useful to fully utilize the potential offered by this promising technology.

References

1. Bernstein, D. J.: Circuits for integer factorization: a proposal, available at http://cr.yp.to/factorization.html#nfscircuit
2. Shamir, A., Tromer, E.: Factoring Large Numbers with the TWIRL Device, Proc. Crypto 2003, LNCS 2729, Springer-Verlag, 2003, available at http://www.wisdom.weizmann.ac.il/~tromer/
3. SRC Inc. Web Page, http://www.srccomp.com/
4. Enge, A.: Elliptic Curves and Applications to Cryptography, Kluwer Academic Publishers, 1999
5. IEEE P1363 Standard Specifications for Public Key Cryptography, November 1999. Draft Version 13
6. Rosing, M., Implementing Elliptic Curve Cryptography, Manning, 1999
7. FIPS 186-2, Digital Signature Standard (DSS), Jan. 27, 2000, available at http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
8. Gura, N. et al.: An End-to-End Systems Approach to Elliptic Curve Cryptography. Proc. CHES 2002, LNCS 2523, (2002) 349-365
9. Hankerson, D., Hernandez, J. L., and Menezes A.: Software Implementation of Elliptic Curve Cryptography over Binary Fields. Proc. CHES 2000, LNCS 1965, (2000) 1-24
10. Wolkerstorfer, J.: Dual-Field Arithmetic Unit for GF(p) and GF(2^m). Proc. CHES 2002, LNCS 2523 , (2002) 500-514
11. LiDIA. A library for computational number theory, Technical University of Darmstadt. Available from http://www.informatik.tu-darmstadt.de/TI/LiDIA/Welcome.html