A Reconfiguration Hierarchy for Elliptic Curve Cryptography

Patrick Schaumont, Ingrid Verbauwhede
Electrical Engineering Department
University of California at Los Angeles

Abstract

Embedded cryptographic applications with tight security and performance constraints require domain specific processors or co-processors. This contribution describes the design model of an elliptic curve public key encryption processor and investigates the design automation requirements of such a processor.

1. Introduction

Providing privacy, authentication and security in general is one of the main concerns for embedded information infrastructure. Electronic encryption and decryption, two fundamental operations in security algorithms, rely on highly specialized operators and algorithms using finite fields and large integer arithmetic. These operations are well suited for implementation in a domain specific processor.

A domain specific processor is a programmable architecture that has primitives tuned to one particular application domain. Such a processor typically augments the standard software environment of an embedded system. It is well known that specialized architectures outperform general purpose ones in terms of power consumption and/or throughput cost factors [1].

In this contribution, we argue that a domain specific security processor contains multiple levels of programming, and thus is not simply programmed by a closed format language such as C. The reasons for this are

1. The domain specializations are not well supported by general purpose programming languages. Instead, a use model based on API is much more effective.
2. In order to justify a domain specific processor, flexibility is of prime importance. Therefore we want to create a parametrizable device that can support a class of architectures and applications rather then a single one.

Secure processing also puts severe constraints to the accessibility of security engines and the data they process (e.g. private keys). Software, which operates in a shared memory environment, is fundamentally unsecure and hard to protect. A domain specific processor on the other hand, can be to made operate in a separate data space and, by careful design of access, resistant to attacks [2].

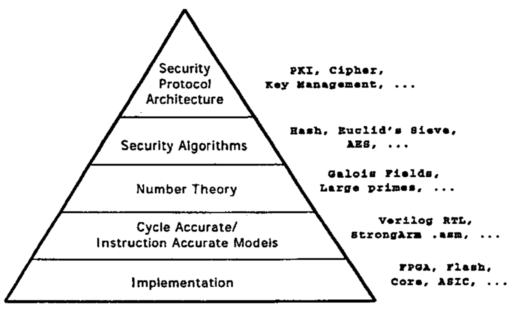
Security Protocol Architecture
Security Algorithms
Number Theory
Cycle Accurate/ Instruction Accurate Models
Implementation
PKI, Cipher, Key Management, ...
Hash, Euclid's Sieve, AES, ...
Galois Fields, Large primes, ...
Verilog RTL, StrongArm .asm, ...
FPGA, Flash, Core, ASIC, ...

Figure 1 Security Pyramid

The organization of this paper is as follows. In section 2, we briefly overview the cryptographic domain and how it relates to embedded processing. In the next section we discuss the design of an elliptic curve encryption processor. We also illustrate the reconfiguration hierarchy principle (multiple levels of programming) on this architecture. The next chapter enumerates the requirements for design technology of such cryptographic processors, and introduces a design description language that helps designing them. Finally conclusions are drawn.

2. Cryptographic Domain

The properties of the cryptographic domain affect processor design in very peculiar ways. This is clarified in this section.

2.1. The security pyramid

We first briefly present an engineer's view on the application domain in the form of a security pyramid, as shown in figure 1. The pyramid form represents the design space at multiple levels of abstraction [3].

The most abstract representation of a cryptographic application is the security protocol architecture, which details what steps make up a secure communication. Examples are IPSEC, SSL, WEP, etc. This covers aspects such as key management and distribution, as well as the placement of cipher blocks within the information flow of a complete application. At this level, an encryption processor looks like a single box that takes care of the implementation of one or more steps in the overall security protocol. A security protocol itself is described usually in plain text format, for example [4].

The next level represents the security algorithms. An example of an encryption algorithm is Rijndael, the recently selected AES standard [5]. A security algorithm is specified by the combination of a signal flow graph to express the data operations, in combination with some overall control sequencing like e.g. feedback modes of operation.

The operations used as cryptographic building blocks are derived from number theory and make up the next level. Besides the operations, also the number representations are specific. For example, in the normal basis of the Galois field GF(2^p), elements are represented as binary coefficients of a polynomial.

Beyond the level of number theory we run into levels that deal with implementation issues. Contemporary embedded platforms express behavior in terms of cycle-accurate and/or instruction-accurate code. This code is mostly platform independent. At this level the algorithm state has been mapped to a storage hierarchy, so a rather detailed allocation of storage has been done.

Finally, at the bottom level we express all aspects of a security algorithm in terms of target platform technology.

It is seen that lower modeling levels become more generic and thus can potentially be shared with other application domain pyramids. For example, reed solomon block coding, used in channel coding, uses galois field operations and thus can share all levels of the security pyramid up to the number theory. When building a domain specific processor, it is important to keep overlap with related domains in mind.

2.2. Properties of cryptographic processors

The nature of the security pyramid puts very specific requirements to the implementation of cryptographic processors. We have to consider the following issues.

1. The large wordlengths found in typical finite fields (1024 bit for RSA) require wide, bit-sliced data-paths. Bit-slicing is mandatory to maintain hardware synthesis quality. On the other hand, writing bit-slicing in HDL is a tedious and non-trivial task.
2. Feedback is a fundamental mode of operation for some cipher operations. Pipelining is not an effective option to obtain performance improvement in those cases [6].
3. Number representation is non-standard and can even take on several different styles within the same cryptographic processor [7]. This is because the cost of operators varies widely with the particular number representation.

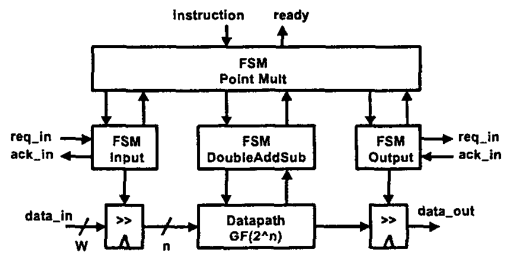

Figure 2 Elliptic Curve Encryption Processor

4. Integration requires special attention if security is not to be compromised. This includes the use of a well-defined and well behaving data and control interface (API), as well as maintaining strict isolation of internal processing to eavesdroppers and less friendly attackers.

3. Elliptic Curve Processor

Figure 2 shows the architecture of an elliptic curve encryption processor [8] that calculates keys for the current IEEE public-key encryption standard [9]. We briefly explain the principles of public key cryptography and next discuss the architecture in more detail.

3.1. Public Key Cryptography

Elliptic-curve public key cryptography is based on the operations on points of a specific curve in a finite field, the so-called underlying field. The point multiplication is the fundamental operation for the key agreement protocol. The Diffie-Hellman key agreement protocol works as follows [10]: given a point P on the curve, Alice will compute *a.P*, and Bob will compute *b.P*. Alice receives *b.P* and computes *a.b.P*. Bob receives *a.P* and computes *a.b.P*. They now share a secret key *a.b.P*. The assumption is that an eavesdropper, who has access to *a.P* and *b.P* cannot compute *a.b.P* because the discrete logarithm problem is a hard problem in the elliptic curve group. The algorithm can be implemented across different abstraction levels. At the highest level, the point multiplication k.P is executed, where k is an integer and P is a point on the elliptic curve. The point multiplication can be decomposed into doublings, additions and subtractions of points on the elliptic curve.

These primitive operations on points of the elliptic curve can again be decomposed in operations on elements of the underlying field. These operations are the addition, the multiplication and the squaring of elements of the underlying field.

3.2. ECC Processor

The architecture of figure 2 has a layered structure, with the layers corresponding to the operation described above.

- A Galois Field datapath implements addition, squaring and multiplication of elements of an n-bit Galois field in normal basis.
- The FSM DoubleAddSub implements the basic elliptic curve operations that are needed for a point multiplication. DoubleAddSub will translate those operations into Galois Field additions, squarings and multiplications.
- The FSM PointMult implements the top-level sequencing of the point multiplication, and also presents a user API in the form of an instruction set as shown in Table 1.
- The FSM Input and Output implements data-IO, and adapts the host system buswidth to the internal ECC processor buswidth.

Both the control interface (at FSM PointMult) and the data interface (at FSM Input and Output) are supported by two-way handshakes. This allows easy integration of the ECC processor in a system, and even allows it to run at an unrelated clock.

Table 1: ECC Instruction Set

Instr	**Opcode**	**Description**
SETP	0001	Set Irreducible Polynomial
SETA	0010	Set EC parameter a
SETB	0011	Set EC parameter b
SETN	0100	Set Point Multiplier n
SET3N	0101	Set Point Multiplier 3n
SETX	0110	Set Initial Point X
SETY	0111	Set Initial Point Y
PMLT	1000	Point Multiplication
PMLN	1001	Point Multiplication and Negate
GETX	1010	Readout X
GETY	1011	Readout Y
GETZ	1100	Readout Z
	Default	Nop

3.3. Programming the ECC processor

The ECC architecture has several different parameters that need to be programmed using the instructions of table 1 before point multiplications can be performed. First, the elliptic curve must be uniquely defined. An elliptic curve over GF(2^n) is given by

$$y^2 + xy = x^3 + ax^2 + b$$

Parameters a and b must be chosen (SETA, SETB). The points on this curve are elements of a finite field GF(2^n). This field is defined by an irreducible polynomial p that has to be selected as well (SETP). During operation, one presents an initial point (X,Y), sets the multiplicand integer n and starts the point multiplication (SETX,SETY,SETN,PMLT). When this last instruction ends, one can read out the resulting point (X,Y,Z) in projective coordinates (GETX,GETY,GETZ).

Depending on the security protocol architecture, also other elements can be required to vary. For example, increasing the finite field size enhances the encryption speed but at the same time also the cipher strength. Finite field size can be made reprogrammable by varying the number of active bitslices in the data-path.

4. Design Aspects

In order to describe this design efficiently, we have build a language and simulation environment that allows us to explore the design of such domain specific processors at a high abstraction level, but without losing the link to automatic implementation. As we will show, one of the harder aspects in the design of the ECC processor is the control architecture. The hierarchy of FSM introduces complex handshaking mechanisms in order to maintain synchronization.

4.1. Datapath

We focus on one particular operation out of the ECC datapath, which is Galois field multiplication. A bit-

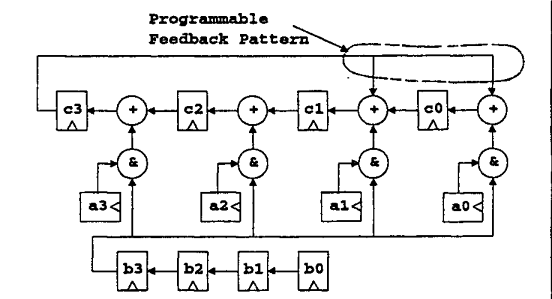

Figure 4 GF(2^4)multiplier for g(x) = x^4+x+1

serial multiplication, not yet bitsliced, is shown in Figure 4. This flowgraph multiplies bitvectors a and b, both in GF(2^4) representation to yield bitvector c. Arithmetic in GF(2^4) is governed by a field polynomial which is selected by the feedback pattern of the structure. Listing 1 shows a textual representation of the same structure. The datapath that is created has a set of state registers (`reg` variables) that are subject to expressions within a signal flowgraph `sfg`. An `sfg` represents one clock cycle of processing, thereby making this a clock cycle true description. The `sfg` uses word-level semantics, which allows compact descriptions. The structure uses also a local one-hot controller (`ctl`), counting the 4 clock cycles the bit-serial structure needs to complete.

Listing 1: Bit-serial multiplier in GF(2^4)

```
dp D( in a, b : ns(4); out mul: ns(4);
      in mul_st: ns(1);
      out mul_done : ns(1)) {
 reg ctl, cr, br, ar : ns(4);

 sfg s1 {
  ctl       = mul_st ? 1 : (ctl << 1);

  ar = a;
  br = ((ctl==0) ? b : (br << 1));
  cr =  (ctl==0) ? 0 : (cr << 1)
                     ^ (ar & (tc(1))  br[3])
                     ^ (0b0011 & (tc(1)) cr[3]);
  mul = acc;
  mul_done = ctl[3];
}}
```

Listing 1 implies allocation of datapath resources since all operations execute in the same clock cycle and thus require parallel implementation. By allowing multiple `sfg` instances per datapath (instructions), and introducing a separate controller description in the form of a sequencer or a finite state machine, we obtain a description that also supports operator sharing. This is demonstrated in listing 2.

While the description in listing 2 is bigger then that of listing 1, it obtains a better separation of concerns [11] then before. Control is described with a finite state machine model that expresses datapath `sfg` execution on the state transitions.

4.2. Simulation

The descriptions in Listing 1 and 2 can be parsed in to yield an object hierachy (in C++), as shown in figure 5. This object hierarchy next can be analyzed by a simulation kernel or a code generation kernel for the purpose of cycle-true simulation and HDL code

Listing 2: FSM-controlled bit-serial multiplier

```
dp D( in a, b : ns(4); out mul: ns(4);
      in mul_st: ns(1);
      out mul_done : ns(1)) {
 reg ctl, cr, br, ar : ns(4);
 sfg ini {
  ar = a;  br = b;
 }
 sfg calc {
  cr = (cr << 1) ^ (ar & (tc(1))  br[3]) ^
       (0b0011 & (tc(1)) cr[3]);
 }
 sfg outactive {
  mul      = cr;  mul_done = 1;
 }
 sfg outidle {
  mul      = 0;   mul_done = 0;
 }
}
fsm F(D) {
  state s1, s2, s3, s4, s5;
  initial   s0;
  @s0 (ini,  outidle) -> s1;
  @s1 if (mul_st_cmd) then (calc,outidle)-
>s2;
                       else (ini, outidle)-
>s1;
  @s2 (calc, outidle) -> s3;
  @s3 (calc, outidle) -> s4;
  @s4 (calc, outidle) -> s5;
  @s5 (ini,  outactive) -> s1;
}
```

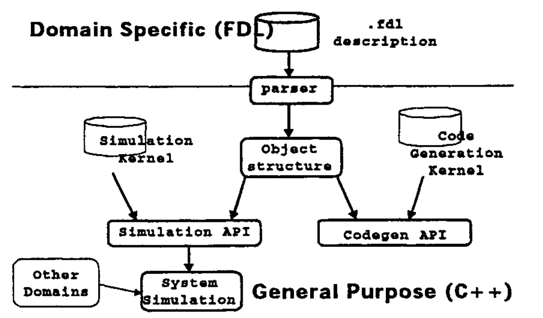

Figure 5 Design Automation Infrastructure

generation respectively. The kernels are presented to the user through a simple C++ API. A system simulation then consists of writing a C++ program and calling the parsing and simulation API as needed to execute the domain specific processor.

This approach clearly distinguishes between a domain specific part, written in a domain specific language, and a general purpose part in C++. As such, it is a meet-in-the middle approach between general purpose approaches such as OCAPI [12] or SystemC, and language specific approaches such as SpecC [13]. This setup allows a designer, being expert in a particular domain, to use descriptions that are concise with the domain semantics. At the same time, the descriptions can be easily linked into a system simulation, where different design domains are combined. We do believe that domain specific processing presents an area where higher abstraction levels can be developed easier then for the generic 'system design language'case.

The most simple use model of figure 5 is where we build one generic system simulation model that parses in a design description and simulates it. In that case we

Listing 3: Testbench for multiplier

```
// testbench
dp TB( out i1, i2 : ns(4); out mul_st: ns(1))
{
  reg ctl : ns(4);
  sfg s1 {
    ctl = ctl + 1;
    i1     = 0b1101;
    i2     = 0b1001;
    mul_st = (ctl == 4) ? 1 : 0;
  }
}
hardwired F2(TB) {s1;}

system S {
  D  (fp, i1, i2, mul, mul_st, mul_done);
  TB (fp, i1, i2, mul_st);
}
```

Listing 4: Generic System Simulation Model

```
#include <fdlsim.h>
int main(int argc, char **argv) {
  symbolTable table = call_parser(argv[1]);
  rtsimgen simulator;
  table.create_simulator(simulator);
  simulator.run(atoi(argv[2]));
  return 0;
}
```

design also a testbench in the same domain language. An example testbench is shown in listing 3. The testbench is described in the same datapath semantics as used for the multiplier. A `hardwired` controller is used because TB always executes the same instruction `s1`. Finally, a `system` statement is used to connect the testbench to the GF multiplier of Listing 2.
The generic system simulation model that parses the testbench and the multiplier is a small C++ program as shown in Listing 4.

4.3. Hierarchical Control

The complete ECC processor consists of several instruction decoding engines on top of each other. These are implemented as hierarchical finite state machines that are mutually synchronized. The example in figure 6 shows two finite state machines out of the ECC processor on figure 2. The left one is a part of DoubleAddSub, while the right one is a part of the GF(2^4) datapath. One particular step of the key setup ECC algorithm is to multiply a coefficient B by itself N times, with both B and N depending on algorithm parameters. The left FSM of figure 6 shows the activity at the DoubleAddSub level. After the parameters are programmed (state 0 and 1), this FSM enters a double nested loop (state 2 and 3), with the inner loop performing bitserial multiplications and the outer loop counting down the N iterations. The right FSM of figure 6 shows the bitserial multiplication control.
While the operation of figure 6 is easily explained, capturing it in state diagrams and subsequently in code is not. This is because figure 6 does not express the real control hierarchy (two nested loops), but rather the way in which it is implemented in finite state machines. Those finite state machines are always active and thus need be be kept synchronized. Effective description of the control hierarchy might be obtained using StateCharts [14] or Esterel [15]. However, for our application domain we observed that no environment currently is available that combines effective datapath description with hierarchical control concepts like exceptions and behavioral completion [13].

5. Conclusions

In this contribution we have presented design motivations behind an encryption processor for cryptography. We presented an hardware implementation of this design. The concept of reconfiguration hierarchy was demonstrated on the design of this processor. Additionally, a design language was presented that helps us in creating these domain specific processors. At this moment, we are investigating mechanisms to express and implement control hierarchies in more compact form as can be done with finite state machines.

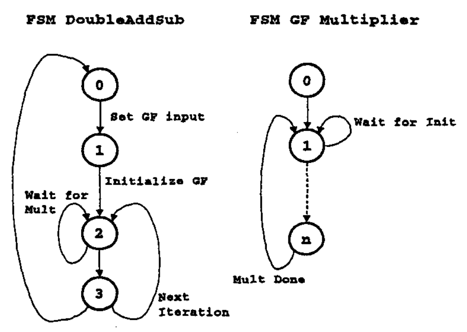

Figure 6 Design Automation Infrastructure

6. References

[1] J. Rabaey, *Silicon platforms for the next generation wireless systems-what role does reconfigurable hardware play?*, FPL 2000, LNCS 1896, Springer-Verlag, August 2000, pp.277-85.
[2] J. Dyer, M. Lindemann, R. Sailer, L. van Doorn, S. Smith, S. Weingart, *Building the IBM 4758 Secure Coprocessor*, IEEE Computer, October 2001, pp. 57-66.
[3] B. Kienhuis, *"Domain Space Exploration of Stream Based Architectures for Dataflow Applications"*, PhD thesis, TU Delft.
[4] http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-09.txt SSH Protocol Architecture July 20, 2001.
[5] The Advanced Encryption Standard, http://csrc.nist.gov/encryption/aes
[6] R.J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, Boston, 1987.
[7] A. Elbirt, W. Yip, B. Chetwynd, C. Paar, *An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists*, IEEE Trans on VLSI, August 2001, pp. 545-557.
[8] S. Janssens, J. Thomas, W. Borremans, P. Gijsels et al, *Hardware-Software Codesign of an elliptic curve public-key cryptosystem*, Proc. SIPS 2001, Antwerpen.
[9] IEEE 1363-2000 Standards for Public-Key Cryptography.
[10] E. Dewin, B. Preneel, *Elliptic Curve Public-Key Cryptosystems: An Introduction*, LNCS 1528, Springer-Verlag, June 1997, pg. 131-141
[11] Keutzer, K.; Newton, A.R.; Rabaey, J.M.; Sangiovanni-Vincentelli, A. *System-level design: orthogonalization of concerns and platform-based design.* IEEE Trans. on CAD, vol.19, (no.12), IEEE, Dec. 2000. p.1523-43
[12] http://www.imec.be/ocapi
[13] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, MA.
[14] D. Harel, *Statecharts: A visual formalism for complex systems*, Sci. Comput. Programming 8, 1987, pp. 231-74.
[15] G. Berry, *The Foundations of Esterel*, Proof, Language and Interaction: Essays in Honour of Robin Milner, *MIT Press*, 2000