Design Automation for Embedded Systems, 7, 365–383, 2002.

Domain Specific Tools and Methods for Application in Security Processor Design

PATRICK SCHAUMONT schaum@ee.ucla.edu
UCLA Department of Electrical Engineering, Los Angeles, CA

INGRID VERBAUWHEDE ingrid@ee.ucla.edu
UCLA Department of Electrical Engineering, Los Angeles, CA

Abstract. Security processors are used to implement cryptographic algorithms with high throughput and/or low energy consumption constraints. The design of these processors is a balancing act between flexibility and energy consumption. The target is to create a processor with just enough programmability to cover a set of algorithms—an application domain. This paper proposes GEZEL, a design environment consisting of a design language and an implementation methodology that can be used for such domain specific processors. We use the security domain as driver, and discuss the impact of the domain on the target architecture. We also present a methodology to create, refine and verify a security processor.

Keywords: Cryptography, domain specific, processor.

1. Domain Specific Processing

1.1. Security Processing Using a Domain Specific Processor

Security processors are used in information infrastructure to improve the implementation of security algorithms. Typical applications include creation of public keys, authentication, bulk data encryption, pseudorandom number generation and digital signatures. The key in domain specific processing is to trade flexibility for power consumption and/or speed. An example of what can be gained from this is given in Table 1. The table shows power consumption and throughput for three different implementations of the same AES algorithm. AES is the new encryption standard [24] that was selected by the NIST in November 2001. The implementations shown in the table use either a domain specific architecture created with standard cells [19], an FPGA or a Pentium-III processor. The table illustrates that the overall figure of merit for a domain specific architecture is more than three orders of magnitude better than that of a general purpose architecture. While general-purpose processing architectures and fine-grain reconfigurable architectures are able to keep up with performance, they loose at length with respect to power consumption.

In order to achieve this result, the programmability of the AES processor has been reduced to the strict minimum. It contains a wide datapath that implements an entire AES iteration as a single instruction. The instruction set of the entire processor has been reduced to 12 instructions, including load_key, load_data, set_key_length, set_data_length, and encrypt. This instruction set covers the basic Rijndael encryption algorithm.

Table 1. AES Implementation on Three Different Platforms

Platform	Power (W)	Throughput Gb/s	Figure of Merit (Gb/s/W)
Domain specific processor[1]	0.056	1.62	28.9
FPGA[2]	0.490	1.323	2.7
Pentium-III, 1.13 GHz[3,4]	41.4	0.648	0.015

1. 0.18 μm CMOS standard cell design of Rijndael [19]
2. Amphion CS5230 on Virtex-II, Power Estimator XAPP152
3. http://www.cs.tut.fi/helger/rijndael.html
4. Power based on Intel Datasheet (Vcc = 1.8 V, Icc = 23 A)

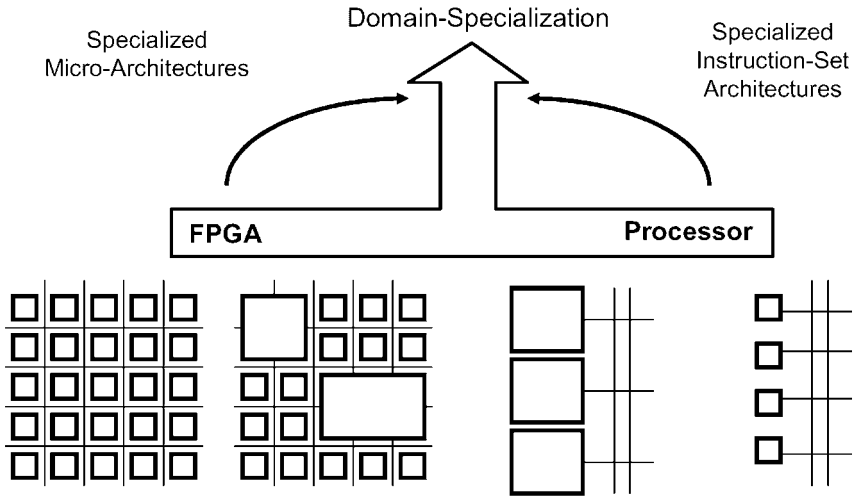

Figure 1. Domain specialization reduces programmability and/or reconfigurability.

1.2. The Design Problem

The dramatic improvements that can be obtained with domain specialization require careful reduction of the programmability and/or reconfigurability of a general purpose architecture into a domain specific architecture. As shown in Figure 1, this is a dual-ended problem. Given a fine-grain reconfigurable architecture like an FPGA, we need to find the micro-architecture specializations that span an application domain. These can be custom operators, routing resources, storage architectures or combinations thereof. Alternatively, we might start from a general purpose processor and introduce custom instruction-set extensions that are well suited for the application domain. Thus, the design space for domain specialization is vast and difficult to conquer. In fact, there are numerous commercial offerings and academic platforms available that elaborate the idea of reducing programmability in favor of domain specialization [3], [25], [11].

1.3. Outline

In this paper we present a design methodology and a language, called GEZEL, for the creation of security processors. We will first give an overview of electronic security and the related design domain. We introduce the concept of reconfiguration hierarchy as a design space in which domain-specific programming is done. In Section 3, we list the architecture properties of typical security processors and discuss an example processor, used in elliptic curve processing. In Section 4, we combine the ideas of Sections 2 and 3 into a design method and a design flow. To express the domain-specific parts (the security processors), we introduce the GEZEL design data model, based on an extension of the Finite State Machine and Datapath (FSMD) model [6]. The FSMD model expresses behavior as a combination of a finite state machine that controls a datapath. In addition, a language to express designs with this data model is presented. In Section 5 we compare and contrast our approach to related efforts, and then indicate some pending challenges in the conclusions section.

2. Domain-Specific Processors for the Security Domain

2.1. The Security Pyramid

Figure 2 presents an engineer's view on the application domain in the form of a security pyramid. The pyramid form represents the design space at multiple levels of abstraction [16]. The most abstract representation of a cryptographic application is the security protocol architecture, which defines 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 an 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, see for example [14].

The next level represents the security algorithms. An example of an encryption algorithm is Rijndael which was used to define the recently selected AES standard [24]. A security algorithm is specified by 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 in these algorithms 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. Below 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. Finally, at the bottom level we express all aspects of a security algorithm in terms of target platform technology. It can be seen that modeling at lower abstraction levels is 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.

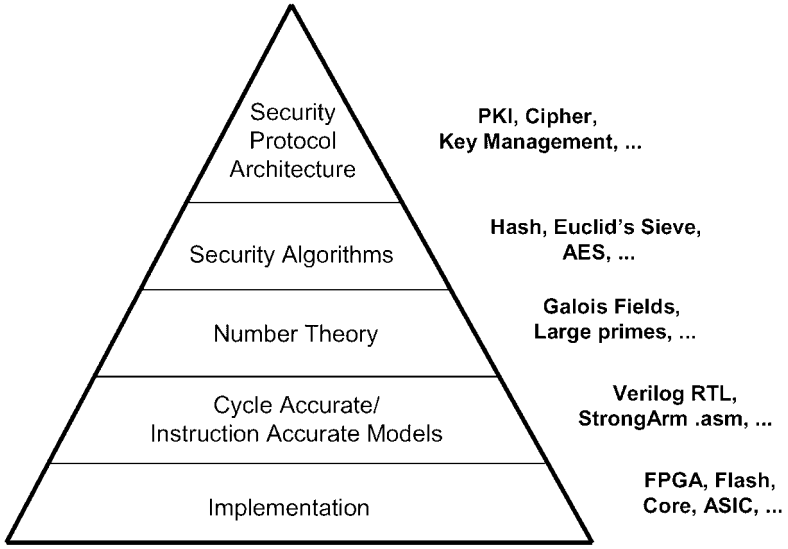

Figure 2. Security pyramid

2.2. Programming in the Security Pyramid

Since reducing programmability is a key aspect in doing domain specialization, it is worthwhile to define programmability and reconfigurability in terms of the security pyramid.

During the design of a domain specific processor, we find out how much programmability and reconfigurability is needed at each level of the design abstraction. For example, full AES encryption can be done with 128-bit, 192-bit or else 256-bit blocks, and therefore full AES has at least three configurations at the level of a security algorithm. AES-128 on the other hand, restricts the encryption block-length to 128 bits. Clearly an AES-128 processor can be implemented more energy-efficient than a full AES processor. This is so because we can propagate these invariabilities down to lower abstraction levels.

When we express a domain specific processor in terms of the security domain pyramid, we see that it spans part of the domain and that it covers a smaller pyramid within the overall one. This is illustrated in Figure 3, that shows the design abstraction levels of an AES processor. At the highest level of abstraction, the AES is a single node in a security protocol architecture that reads in blocks of plaintext data and writes out blocks of ciphertext. At this level, configurability is restricted to a few simple parameters, for instance choice of the encryption key, or data block length. Going one level down in abstraction level, we can see that the AES uses a $GF(2^8)$ inversion and several different matrix transformations. Because algorithm internals like finite field representation are fixed by the AES standard, there is actually no configuration to do at this level. At the next level however, where we decide on the implementation platform, the question of programmability is back. The single function node at security algorithm level is decomposed, either spatially or temporally. We could for instance opt for a micro-programming approach, in

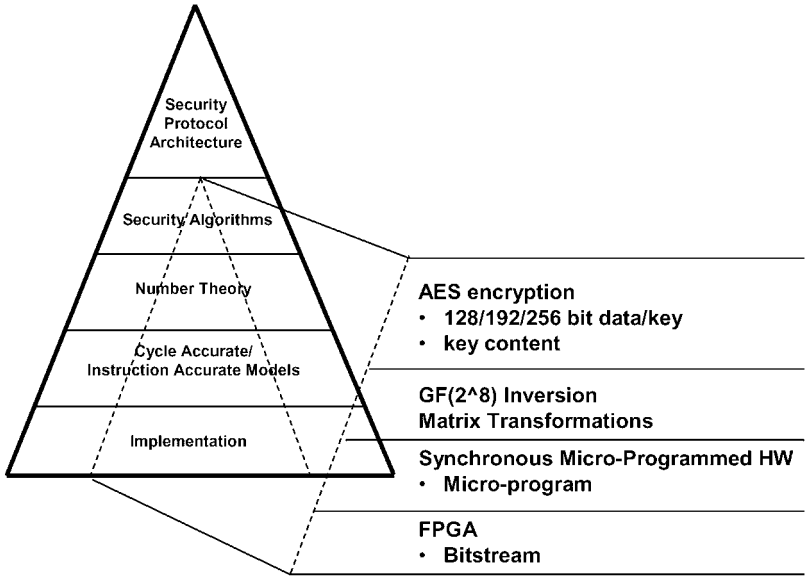

Figure 3. AES processor within the security pyramid.

which case the micro-program defines the configurability of this level. Finally, also the decomposed form of the AES is mapped onto an implementation platform that can be reprogrammable as well, for example a bitstream-programmed FPGA.

2.3. Reconfiguration Hierarchy

This AES processor example leads us to observe that a domain-specific processor covers several different, hierarchical levels of operating abstraction, where potential configurability is decreasing from bottom to top. Domain specialization is obtained by determining, at each level of abstraction, how to reduce the programmability to the strict minimum. In order to reason about this problem, we introduce a design space of programmability and call this a reconfiguration hierarchy [27]. The reconfiguration hierarchy design space has three independent axes.

— A vertical axis that expresses the level of processing abstraction.

— A horizontal axis that expresses the reconfigurable feature diversity.

— A time axis that expresses the timing relationship of configuration to processing.

The vertical axis is related to the level of computation abstraction. At the lowest level we naturally recognize logic primitives (gates), simple storage (registers) and routing. At

Table 2. The Reconfiguration Hierarchy

	Communication	**Storage**	**Processing**
System	Interconnection Network	Buffer Size	Number and Type of asynchronous processes and tasks
Instruction Set	Address / Data Buswidth	Register Set Memories	Custom Instr Interrupt Levels
Micro-Architecture	Bus Topology	Register File	EXU Type
Implementation			Interpreter Levels
Circuit	Mux and Switch Interconnect	RAM Org Latch Transparency	LUT

higher levels, the microarchitecture, instruction-set architecture and process architecture (or systems architecture) represent additional layers of computation abstraction. Reconfiguration is applicable to each new abstraction layer that is introduced. While the vertical axis describes a hierarchy, the horizontal axis describes the nature of reconfigured elements. Each level of the hierarchy is made up of a combination of communication, storage, computation and control. Reconfiguration can affect each of those individually. In Table 2, an enumeration is given of different such design elements (horizontal) characterized at different design levels. The term coarse grain and fine grain reconfigurability are usually associated with the variation of horizontal features at the architectural level. The binding time expresses when configuration data is send to the processing part. Each level of the hierarchy can be bound individually. We distinguish implementation-time binding and design-time binding. With implementation-time binding, configuration is postponed until actual execution of the processing part is required. With design-time binding, configuration is done at the moment that the processing part is conceived. This is equivalent to hard-coding. These terms are preferred over the more traditional run-time and compile-time since the latter ones are not unique for hierarchical systems. In order to have a physical implementation, the lowest processing level of a system is always design-time bound. The top level of programmable systems is always implementation-time bound. In between, there is a smooth transition called the binding time continuum [3].

3. Security Processors

3.1. Properties of Security Processors

Security processors have specific architectural features which are enumerated here.

— The large wordlengths found in typical finite fields (1024 bit for RSA) require wide, bit-sliced data-paths. Bit-slicing helps to maintain hardware synthesis quality.

— Multitiered control structures naturally support the hierarchy of behaviors that is present in security algorithms and protocols [8], [15]. They allow to reflect the

security pyramid in the architecture and enable precise control over which part of a processor is programmable and which is not.

— Feedback is a fundamental mode of operation for some cipher operations. Pipelining is not an effective option to obtain performance improvement in those cases [31].

— Number representation is non-standard and can even take on several different styles within the same cryptographic processor [4]. This is because the cost of operators varies widely with the particular number representation.

— Specialized arithmetic operations such as Modular Arithmetic and Galois Field Arithmetic require specialized operators [22].

— For block-mode ciphers, the input−output structure is block-oriented. In addition, data blocks are typically larger than the host machine word-size.

— Integration requires special attention if security is not to be compromised [5]. This includes the use of a well-defined and well behaved data- and control interface (API), as well as maintaining strict isolation of internal processing to eavesdroppers and less friendly attackers.

3.2. An Elliptic Curve Encryption Processor

Figure 4 shows the architecture of an elliptic curve encryption processor [15] that calculates keys for the current IEEE public-key encryption standard [13]. We briefly explain the principles of public key cryptography and next discuss the architecture in more detail.

Elliptic-curve public key cryptography is based on operations on points of a specific curve in a finite field, the so-called underlying field. Point Multiplication is the fundamental operation for the key agreement protocol. The Diffie−Hellman key agreement protocol works as follows [4]: 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 $a.b.P$. Due to the properties of the elliptic curve group, it is however very hard to calculate $a.b.P$ starting from the knowledge of $a.P$ and $b.P$ alone. An eavesdropper, who has access to $a.P$ and $b.P$, can therefore not obtain the common secret key—or at least has to solve the very hard mathematical problem of discrete logarithm in the elliptic curve group.

This 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 addition, multiplication and squaring of elements of the underlying field.

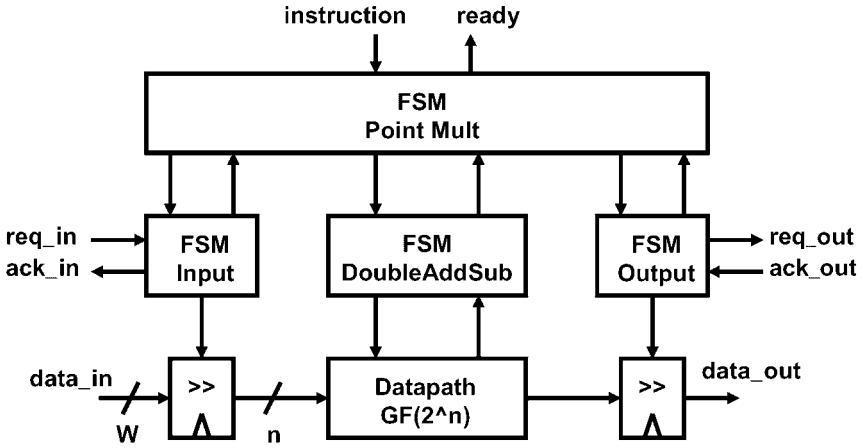

Figure 4. Elliptic curve encryption processor.

The processor that implements this algorithm is an Elliptic Curve processor (ECC), and is shown in Figure 4. This architecture 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 underlying finite field).

— 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 instruction set of DoubleAddSub is shown in Table 3.

— 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 4. The instruction set of FSM DoubleAddSub is shown as well in this table.

— 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 into a system, and even allows it to run at an unrelated clock.

The ECC architecture has several different parameters that need to be programmed using the instructions of Table 4 before point multiplications can be performed. First, the elliptic curve must be uniquely defined. An elliptic curve over $GF(2^n)$ is a curve in two variables x and y that has two parameters a and b [13].

Table 3. FSM Instruction Set for DoubleAddSub

Instr	Opcode	Description
SETP	0001	Set Irreducible Polynomial
SETA	0010	Set EC parameter a
SETB	0011	Set EC parameter b
SETX	0100	Set X into Xreg & T1
SETY	0101	Set Y into Yreg & T2
READX	0110	Readout X
READY	0111	Readout Y
READZ	1000	Readout Z
DBL	1001	Eval $P = 2P$
ADD	1010	Eval $P = P + (X, Y)$
SUB	1011	Eval $P = P - (X, Y)$
INV	1100	Eval $P = -P$
	Default	Nop

Table 4. FSM Instruction Set for Pointmult

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	Set Initial Point X
GETY	1011	Set Initial Point Y
GETZ	1100	Set Initial Point X
	Default	Nop

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 (not to be confused with the point P on the elliptic curve) 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, other elements can be required to vary. For example, increasing the finite field size reduces the encryption speed but at the same time also increases the cipher strength. Finite field size can be made reprogrammable by varying the number of active bitslices in the data-path.

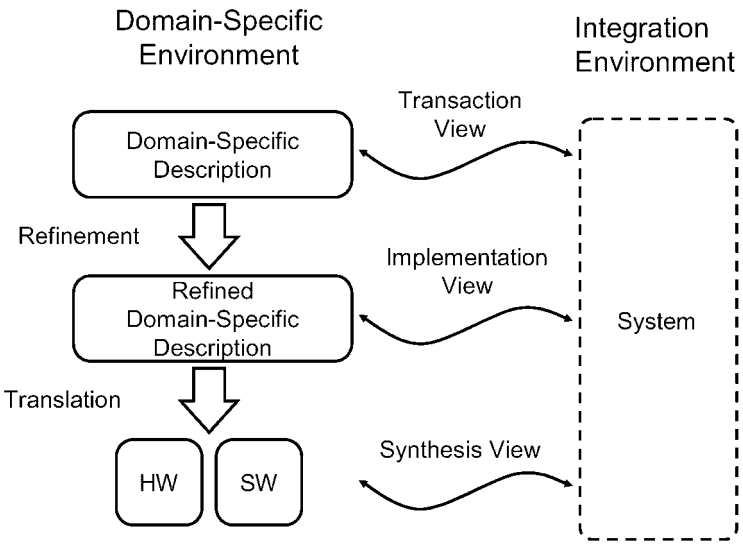

Figure 5. Domain specific design flow.

4. GEZEL Design Environment

We now propose an approach for design automation support of domain specific processors. This consists of a design flow, combined with a design language.

4.1. GEZEL Design Flow

The GEZEL design flow for a domain-specific processor is shown in Figure 5. We use a view that fits the initial (functional) partitioning of a system, and treat a system as a composition of different design domains. When we now concentrate on the design process within a single design domain, we make a distinction between the design of the domain-specific part itself and the integration of this part into the system. Therefore, the design flow in Figure 5 contains a domain-specific side and an integration side. At the domain-specific side, our security processor is designed step-by-step, starting from a high level algorithmic specification and gradually evolving into a detailed architecture description at the cycle-true level. At the integration side, we deal with the problem of how our processor talks to the rest of the system.

A domain-specific design presents a set of views to the integration environment. A view is the ensemble of interactions of a domain-specific part with the integration environment at a particular abstraction level. In Figure 5, three views have been defined as an example.

— A transaction view is a high level co-simulation interface. This could be a transaction based co-simulation interface [28]. For example, when designing a DES encryption processor, a transaction would send a block of data (a transaction) from the system to the DES processor, which would reply with an encrypted version of that block (a second transaction).

— An implementation view is a target-specific co-simulation interface. This could be a clock-cycle true interface description (such as the description of an AMBA bus interface).

— Finally, a synthesis view combines the design result of one domain with the rest of the system. For example, when designing an acceleration unit for an embedded system the synthesis view would be the combination of an HDL description of the processor, together with a device driver and possible other application software that allows integration of the processor.

The domain specific design itself proceeds through refinement and eventually translation to the design target. Refinement varies with the application domain. In the cryptographic domain for example, refinement includes design of security protocol architectures, selection of encryption algorithms, selection of finite field sizes and arithmetic operators, and more.

4.2. GEZEL Implementation Methodology

A rigorous implementation of the concept in Figure 5 is shown in Figure 6. We describe a domain specific processor using a language that is tuned towards the application domain. Such a domain specific language can be lean since it is restricted to one application domain.

This language is parsed and converted into an object structure in a general purpose programming language. Currently we are using C++ for this. This object structure makes use of modeling objects provided by a predefined library. The library provides services such as simulation and code generation, and makes those services available through an application program interface (API). This interface is used to construct a system simulation, and eventually to convert the domain-specific description into synthesizable code.

GEZEL 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 SystemC [28], and language specific approaches such as SpecC [7]. The 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 are fully accessible through the C++ API, which provides access to the simulation and code generation kernel. This way, the domain specific 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 than for the generic system design language case.

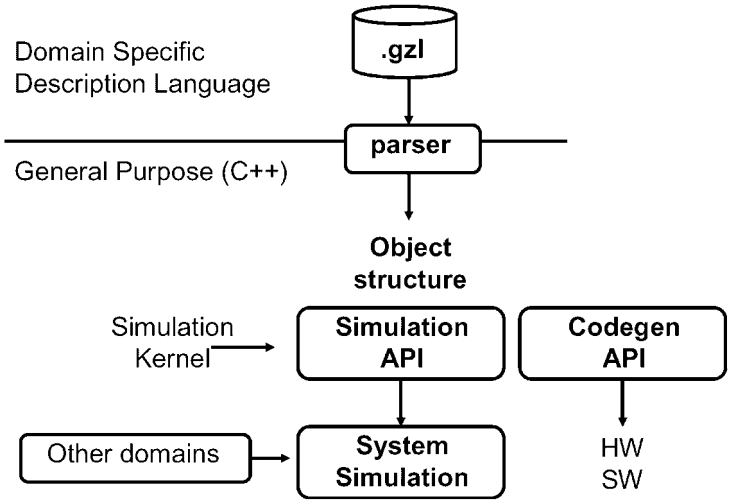

Figure 6. Gezel implementation.

4.3. GEZEL Design Data Model

The GEZEL domain specific language is a representation of a more fundamental structure, a design data model. This design data model is crafted along the requirements of security processors as discussed in Section 3.

We started from a model consisting of communicating FSMD. In such a model, each processor of the system is expressed as the combination of a finite state machine in combination with a datapath. In our model we expressed the datapath at behavioral level as a set of signal flowgraphs (SFG). The controller model decides, at each instruction cycle, which of the datapath SFG to execute. Currently we support three kinds of controller models: the Hardwired Controller (that always execute the same SFG), the Sequencer (that execute a cyclic sequence of SFG) and the generic Finite State Machine (that supports decision making). Part of our current research is to investigate what kind of controller models are best suited for security processors.

Split representation of data-processing and control-processing was found very convenient for modeling at the micro-architecture level. But this approach has been used at other abstraction levels too. Examples at higher level of abstraction are FunState [29] and SBF [17]. The FSMD model has also been proven to be applicable to real designs, for example in OCAPI [30].

4.4. GEZEL Domain Specific Language

Before introducing the GEZEL language, we motivate why we are defining *yet another* language. First, we want to make clear that our goal is not to define a language that can describe a complete system. Rather, we are looking for a way to express the design data model defined above in a convenient way. This leaves us with several different options.

— We can use a general purpose programming language (C++) such as is done in SystemC. This is a very good approach in the initial phases of the creation of a design data model. It allows to modify the design data model at any time, and also provides simple programming as a fallback for those parts of a design where the design data model does not fit. The drawback of this method is that the design environment (C++ compiler) cannot make any distinction between design objects that are part of the design and objects that are part of the design data model. For example, when designing a finite state machine with C++ objects, a designer will still see C++ syntax errors (rather than FSM syntax errors). This requires the designer to become expert in both the design and the design data model.

— We can use a custom graphical syntax, such as a block based model. For some application areas like controller design or dataflow graphs, very good representations are available. The drawback here is that there is no good graphical representation available that captures both the data processing and control processing aspects of our design data model, and as a result we have to use mixed graphical/textual models. Such models are inherently more complex to handle.

— We can use a domain specific language. Such a language is a direct syntax representation of the design data model that fits the domain specific part of the design. Some examples of this approach from the software engineering world are Yacc for the construction of parsers and Perl for text processing.

We will use an example to describe the GEZEL domain specific language. We focus on one particular operation out of the ECC datapath, which is Galois field multiplication. Figure 7 shows a bit-serial multiplication. This flowgraph multiplies bit-vectors a and b, both in $GF(2^4)$ representation to yield bit-vector c. Arithmetic in $GF(2^4)$ is governed by a field polynomial which is selected by the feedback pattern of the structure. The next listing shows a textual representation of the same structure.

```
//--Listing 1
dp D( in a, bo       :  ns(4);
      out mul        :  ns(4);
      in mul_st      :  ns(1);
      out mul_done :  ns(i)) {
  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])
```

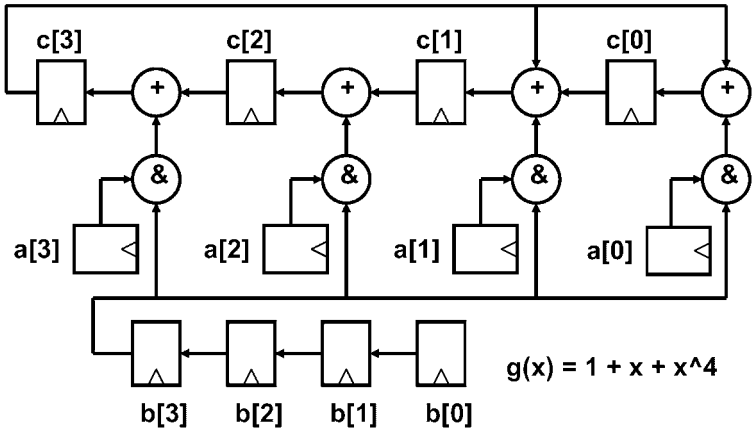

Figure 7. $GF(2^4)$ bitserial multiplier.

```
                    ^(0b0011 & (tc(1)) cr[3]);
  mul = cr;
  mul_done = ctl[3];
  }
}
```

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-parallel semantics, which allows to obtain compact descriptions. The structure also uses a local one-hot controller (`ctl`), counting the 4 clock cycles the bit-serial structure needs to complete. 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 the next listing.

```
// -- Listing 2
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);
  reg mul_st_cmd      : ns(1);
  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; mul_st_cmd = mul_st;
  }
}

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;
}
```

The controller is a FSM with a set of states and transitions between those states. One of these states is the initial one. Conditional transitions are modeled using an if−then−else structure and rely on conditional expressions that are derived from datapath registers. The datapath is now modeled as a collection of `sfg`. Each of the `sfg` `ini`, `calc`, `outactive` and `outidle` represents a single cycle of activity on the datapath. The `sfg` names are used by the controller to define the instruction set.

Comparing listing 1 and 2 we see that separate modeling of datapath processing and control introduces some overhead. On the plus side, the resulting model can express datapath-sharing by defining `sfg` that are executed in exclusive clock cycles.

4.5. Integration

The descriptions in Listings 1 and 2 can be parsed to yield an object hierarchy (in C++), as shown in Figure 6. This object hierarchy can be analyzed by a simulation kernel or a code generation kernel for the purpose of cycle-true simulation and HDL code 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. The most simple use of the model given in Figure 6 is to have a simple testbench in GEZEL together with the design itself.

The system integration task (in C++), in that case, simply executes the GEZEL simulation kernel. An example testbench in GEZEL is shown in the next listing.

```
// -- Listing 3
// 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 (i1, i2, mul, mul_st, mul_done);
    TB (i1, i2, mul_st);
}
```

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 next.

```
// -- Listing 4
#include < fdlsim.h >
int main(int argc, char **argv) {
    // parse GEZEL program
    symbolTable table = call_parser(argv[1]);

    // generate simulator
    rtsimgen simulator;
    table.create_simulator(simulator);

    // run simulation
    simulator.run(atoi(argv[2]));
    return 0;
}
```

4.6. Results for the ECC Processor

We finally compare the energy efficiency of the complete ECC processor to a performance-optimized software implementation [9]. The figure of merit in this case is

Table 5. ECC K-163 Implementation on Two Different Platforms

Platform	Power (W)	Throughput Pmult/s	Figure of Merit (Pmult/s/W)
FPGA[1]	0.575	414	720
Pentium-III, 1 GHz[2,3]	36.6	2600	71

1. [15] on Virtex-II (Xilinx), XST, Power Estimator XAPP152
2. [9], ECSDA with K-163 Curve
3. Power based on Intel Datasheet (Vcc = 1.8 V, Icc = 23 A)

the number of Point Multiplications per second and per Watt. The results are shown in Table 5. The curve parameters were derived from the ECDSA standard [23], that defines standard settings for use in Digital Signature Authentication. We used K-163, which is a setting with an underlying field $GF(2^{163})$. Datapaths thus are 163 bit.

The ECC processor was mapped onto a Xilinx Virtex-II FPGA and occupies 3118 slices with a critical path of 6.5 ns. The low critical path is due to the bit-sliced, bit-serial architecture. The bit-serial design requires 313000 clock cycles on the average per Point Multiplication. This results in 414 Point Multiplications per second. At an estimated power consumption figure of 575 mW, we thus have 720 Point Multiplications per second and per Watt for our processor. The software design yields, despite the higher absolute throughput, only 71 Point Multiplications per second and per Watt.

Based on the results from Table 1, we also note that the figure of merit for a standard cell implementation of the ECC processor would still improve with respect to FPGA.

5. Related Work

Domain specific processors are well known in the domain of signal processing and networking. Our contribution to the field is our focus on architecture design methods, design automation and the related reconfiguration. In particular, we are seeking ways to have an adequate representation of the processing hierarchies in a design.

We have already indicated the relationship to existing system design language research such as SystemC [28] or SpecC [7]. There is an ongoing discussion in the system design community on which language is the right one to use. With this work, we hope at least to conclude that there is a strong dependency of design language to the design domain, and to demonstrate this with real applications.

Domain specific languages have also been used at other abstraction levels. ASIP development environments such as LISA [12] and Chess [20] create both the processor architecture and a software development environment for it out of an instruction-level processor description. In other cases, a domain specific language has also been used to describe the application at behavioral level (rather then architecture- or instruction-level). The Stanford SHADE project [26] uses a dedicated description of shading graphics procedures in order to compile highly optimized code for graphics accelerator hardware.

Heterogeneous co-simulation has also been extensively researched in a variety of environments. We only note Ptolemy [21] and Coware [2] here. Related to this work, we hope to bootstrap on the existing body of research when it comes to selection of the proper simulation mechanisms and models of computation. We indicate the design of a control hierarchy as one of the problems to solve. Existing efforts in this area such as Statecharts [10] and Esterel [1] focus primarily on the modeling of control. We hope to solve the problem of hierarchical control in combination with datapath, connectivity and storage design. And finally, we hope to make efficient use of the software engineering techniques that enable the development of a lightweight language, for which plenty of examples exist [18].

6. Conclusion

In this contribution we have presented GEZEL, an approach for domain specific design based on combining a domain-specific language and a general-purpose language into one environment. The design space for such processors was defined by means of a reconfiguration hierarchy, which recognizes that there are different levels of programming abstraction in a single system that match up against the different levels of design abstraction. The security domain is used as a driver in our approach and we are currently pursuing the development of several demonstrator designs, including a high-speed embedded router with AES support.

References

1. Berry, G. The Foundations of Esterel Proof, *Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
2. Bolsens, I., H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Hardware/Software Co-Design of Digital Telecommunication Systems. In *Proceedings of the IEEE*, vol. 85, no. 3, pp. 391–418, March 1997.
3. Dehon, A. and J. Wawrzynek. Reconfigurable Computing: What, Why, and Implications for Design Automation, *Proceedings of the Design Automation Conference 1999*, June 1999.
4. Dewin, E. and B. Preneel. *Elliptic Curve Public-Key Cryptosystems: An Introduction*. LNCS 1528, Springer-Verlag, June 1997, pp. 131–141.
5. Dyer, J., M. Linemann, R. Perez, L. van Doorn, S. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, pp. 57–67, Oct. 2001.
6. Gajski, D., F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, NJ, 1994.
7. Gajski, D., J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, 2000.
8. Goodman, J., and A. P. Chandrakasan. An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, pp. 1808–1820, Nov. 2001.
9. Hankerson, D. Performance Comparison of Elliptic Curve Systems in Software. In *Proceedings of the Fifth Workshop on Elliptic Curve Cryptography 2001*, Ontario, Oct. 2001.
10. Harel, D. *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Programming, vol. 8, 1987, pp. 231–74.
11. Hartenstein, R. A Decade of Reconfigurable Computing: A Visionary Perspective. In *Proceedings of the Design Automation and Test European Conference 2001*, Munchen, March 2001.

12. Hoffmann, A., A. Nohl, G. Braun, O. Schliebusch, T. Kogel, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Nov. 2001.
13. IEEE P1363/2000: *Standard Specifications for Public Key Cryptography*. http://www.ieee.org.
14. IETF: *SSH Protocol Architecture*. http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-09.txt, July 20, 2001.
15. Janssens, S., J. Thomas, W. Borremans, P. Gijsels, I. Verbauwhede, F. Vercauteren, and B. Preneel. Hardware/Software Co-Design of an Elliptic Curve Public-Key Cryptosystem. In *Proceedings of the 2001 IEEE Workshop on Signal Processing Systems*, pp. 209–216, Antwerpen, 2001.
16. Kienhuis, B. *Domain Space Exploration of Stream Based Architectures for Dataflow Applications* Ph.D. thesis, TU Delft, 1999.
17. Kienhuis, B., and Ed. F. Depettere. Modeling Stream-Based Applications Using the SBF Model of Computation. In *Proceedings of the 2001 IEEE Workshop on Signal Processing Systems*, pp. 209–216, Antwerpen, 2001.
18. Kim, E. The MIT Lightweight Languages Workshop. *Dr. Dobb's Journal*, CMP Publishers, Feb. 2002.
19. Kuo, H., I. Verbauwhede, and P. Schaumont. A 2.29 Gbits/sec, 56 mW Non-Pipelined Rijndael AES Encryption IC in a 1.8 V, 0.18 mm CMOS Technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference 2002*, Orlando, May 2002.
20. Lanneer, D., J. Van Praet, A. Kifli, K. Schoofs,. W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. *Code Generation for Embedded Processors*. P. Marwedel, ed., Kluwer Academic Publishers, 1995.
21. Lee, E. A. Overview of the Ptolemy Project Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.
22. Menezes, A., P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
23. NIST. Federal Information Processing Standards (FIPS) PUB 186-2 Digital Signature Standard. http://www.nist.gov/aes/, Jan. 27, 2000.
24. NIST. Federal Information Processing Standards (FIPS) PUB 197 Advanced Encryption Standard. http://www.nist.gov/aes/, Nov. 26, 2001.
25. Ogrenci, S., E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. SPS: A Strategically Programmable System. In *Proceedings of the Reconfigurable Architectures Workshop 2001*, San Francisco, April 2001.
26. Proudfoot, K., W. R. Mark, S. Tzvetkov, and P. Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of the 28th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*, Los Angeles, 2001.
27. Schaumont, P., I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A Quick Safari Through the Reconfiguration Hierarchy. In *Proceedings of the Design Automation Conference 2001*, Las Vegas, June 2001.
28. Swan, S. An Introduction to System Levtexnansi.encel Modeling in SystemC 2.0, http://www.systemc.org.
29. Thiele, L., K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState—An Internal Design Representation for Codesign. In *Proceedings of the 1999 International Conference on Computer Aided Design*, San Jose, 1999.
30. Verkest, D., W. Eberle, P. Schaumont, B. Gyselinckx, and S. Vernalde. C++ Based System Design of a 72 Mb/s OFDM Transceiver for Wireless LAN. In *Proceedings of the Custom Integrated Circuits Conference 2001*, San Diego, 2001.
31. Whiting, D., B. Schneier, and S. Bellovin. AES Key Agility Issues in High Speed IPsec Implementations. *Public Comments on AES Candidate Algorithms Round 2*, http://www.nist.gov/aes.