An Energy Efficient Reconfigurable Public-Key Cryptography Processor Architecture*

James Goodman[1] and Anantha Chandrakasan[2]

[1] Chrysalis-ITS, Ottawa ON K2C-3R7, Canada
jgoodman@chrysalis-its.com
[2] Massachusetts Institute of Technology, Cambridge MA 02139, USA
anantha@mtl.mit.edu

Abstract. The ever increasing demand for security in portable, energy-constrained environments that lack a coherent security architecture has resulted in the need to provide energy efficient hardware that is algorithm agile. We demonstrate the feasibility of utilizing domain-specific reconfigurable processing for asymmetric cryptographic applications in order to satisfy these constraints. An architecture is proposed that is capable of implementing a full suite of finite field arithmetic over the integers modulo-N, binary Galois Fields, and non-supersingular elliptic curves over $GF(2^n)$, with operands ranging in size from 8 to 1024 bits. The performance and energy efficiency of the architecture are estimated via simulation and compared to existing solutions (e.g., software and FPGA's), yielding approximately two orders of magnitude reduction in energy consumption at comparable levels of performance and flexibility.

1 Introduction

In the past, several standards for implementing various asymmetric techniques have been proposed such as the ISO, ANSI (X9.*), and PKCS standards. The variety of standards[1] has resulted in a multitude of incompatible systems that are based upon different underlying mathematical problems and algorithms. For example, the IEEE P1363 public key cryptography standard [1] recognizes three distinct families of problems upon which to implement asymmetric techniques: integer factorization, discrete logarithms, and elliptic curves.

As a result, system developers have had to either utilize software-based techniques in order to achieve the algorithm agility required to maintain compatibility, or have utilized special purpose hardware and restricted themselves to only providing secure communications with compatible systems. Unfortunately,

* The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

[1] A wise man once said that the best thing about standards is that there are so many to choose from!

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 175–190, 2000.

software-based approaches lead to slow implementations that are very energy inefficient. In the past these inefficiencies could be ignored as the typical user operated from a fixed-location system such as a desktop computer which had a great deal of memory, processing power, and an effectively limitless energy budget. However, with the migration to portable, battery-operated nomadic computing terminals these assumptions break down, requiring us to re-evaluate the use of a software-based implementation. Hardware-based implementations on the other hand, while being very energy and computationally efficient, are very inflexible and capable of supporting only a single type of asymmetric cryptography. A compromise between these two extremes is achieved by taking advantage of the fact that the range of operations is small enough that domain specific reconfigurable hardware can be developed that is capable of implementing the various asymmetric algorithms without incurring the overhead associated with generic reconfigurable logic devices. Furthermore, this is done in an energy-efficient manner that enables operation in the portable, energy-constrained environments where this algorithm agility is required most of all. The resulting implementation is known as the Domain Specific Reconfigurable Cryptographic Processor (DSRCP).

2 Domain Specific Reconfigurability

In conventional reconfigurable applications such as Field Programmable Gate Arrays (FPGAs), the architectural goals of the device are to provide a large number of small, yet powerful programmable logic cells, embedded within a flexible programmable interconnect (e.g., [2], [3]). Unfortunately, the overhead associated with making such a general purpose computing device ultimately limits its energy efficiency, and hence its utility in energy-constrained environments. To illustrate this fact consider the space of all possible functions. A conventional reconfigurable logic device attempts to cover as much of this space as possible given its architectural constraints in terms of technology and logic/routing resources. This results in a considerable amount of overhead that isn't necessary given a specific subset of functions. Kusse [4] quantified this overhead by breaking down the energy consumption of a conventional FPGA (Xilinx XC4003A [5]) into its architectural components (Table 1). The analysis revealed that only 1/20th of the total energy is used to perform useful computation.

Table 1. Energy consumption breakdown of XILINX XC4003A [4].

Component	% Total Energy Consumption
Interconnect	65%
Global Clock	21%
I/O	9%
Logic	**5%**

The DSRCP differs from conventional reconfigurable implementations in that its reconfigurability is limited to the subset of functions, called a domain, required for asymmetric cryptography. This domain requires only a small set of configurations for performing all of the required operations over all possible problem families as defined by IEEE P1363. As a result, the reconfiguration overhead is much smaller in terms of performance, energy efficiency, and reconfiguration time, making the DSRCP feasible for algorithm-agile asymmetric cryptography in energy constrained environments.

3 Instruction Set Architecture (ISA)

The instruction set definition of the DSRCP is dictated by the IEEE P1363 description document. For the various primitives described in the standard, a list of the required arithmetic functions is tabulated in order to determine the required ISA of the processor. Note that certain primitives also require such operations as the ability to set specific bits within a given operand and the ability to generate random bits, neither of which are implemented in this version of the processor. The resulting functional matrix is shown in Table 2.

The functional matrix is used to define the required ISA of the processor, along with additional auxiliary functions for controlling the processor configuration, as well as moving data into, out of, and within the processor. The resulting instruction format for the DSRCP is a 30-bit word partitioned as shown in Figure 1. The DSRCP executes 24 instructions in all, a brief summary of which are given in Table 3.

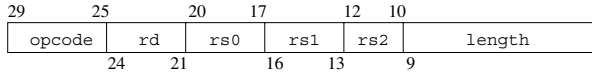

Fig. 1. DSRCP instruction word.

4 Architecture

Figure 2 shows the overall system architecture of the DSRCP. The processor consists of four main architectural blocks: the global controller and microcode ROMs, the I/O interface, the reconfigurable datapath, and an embedded SHA-1 [6] hash function engine. The inclusion of a hash engine was desirable as the key derivation primitives contained within IEEE P1363 call for this functionality.

4.1 Global Controller and Microcode ROMs

The DSRCP features a three-tiered control approach that utilizes both hardwired and microcode ROM-based control functions. This multi-tiered approach

Table 2. Functional matrix of the IEEE P1363 for the DSRCP instruction set.

	ADD	SUB	MULT	MOD	MOD_ADD	MOD_SUB	MOD_MULT	MOD_INV	MOD_EXP	GF_ADD	GF_MULT	GF_SQUARE	GF_INV	GF_EXP	EC_ADD	EC_DOUBLE	EC_MULT
PKO #1									X								
PKO #2			X		X	X	X		X								
IFEP-RSA									X								
IFDP-RSA			X		X	X	X		X								
IFSP-RSA1			X		X	X	X		X								
IFVP-RSA1									X								
DLSVDP-DH														X			
DLSVDP-MQV	X				X		X				X			X			
DLSP-DSA				X	X		X	X									
DLVP-DSA				X			X	X						X			
ECSVDP-DH																	X
ECSVDP-MQV					X		X								X		X
ECSP-DSA				X	X		X	X									
ECVP-DSA				X			X	X							X		X
MOD_EXP					X		X										
GF_EXP											X	X					
EC_ADD										X	X	X	X				
EC_DOUBLE										X	X		X				
EC_MULT															X	X	

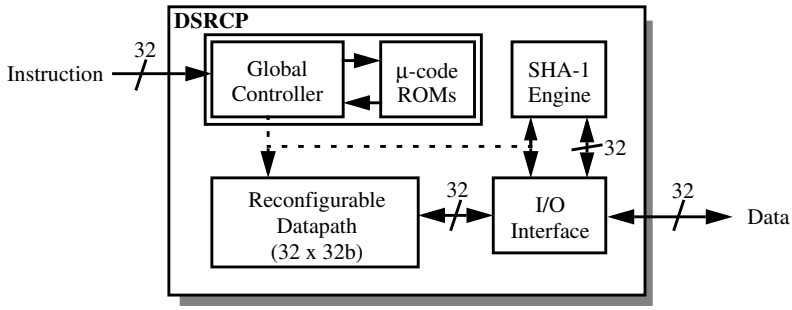

Fig. 2. Overall system architecture of the DSRCP.

Table 3. DSRCP instruction set.

Mnemonic	Description
SET_LENGTH length	sets width of processor to be length + 1
REG_CLEAR rd,rs0	clears registers specified in mask formed by (rd,rs0) = $R_{7:0}$
REG_MOVE rd,rs0	rd = rs0
REG_LOAD rd	rd is loaded from I/O interface
REG_UNLOAD rs1	rs0 is unloaded to I/O interface
COMP rs0,rs1	sets gt = (rs0 > rs1) and eq = (rs0 == rs1) flags
ADD/SUB rd,rs0,rs1,rs2	$rs2_{2:1}$ = 00: rd = rs0 + rs1 + $rs2_0$ $rs2_{2:1}$ = 01: rd = (rs0 + rs1 + $rs2_0$)/2 $rs2_{2:1}$ = 10: rd = rs0 - rs1 $rs2_{2:1}$ = 11: rd = (rs0 - rs1)/2
MOD_ADD rd,rs0,rs1,rs2	rd = (rs0 + rs1 + $rs2_0$) mod N
MOD_SUB rd,rs0,rs1	rd = (rs0 - rs1) mod N
MONTRED_A	$(Pc, Ps) = A \cdot 2^{-n} \bmod N$
MONTMULT	$(Pc, Ps) = A \cdot B \cdot 2^{-n} \bmod N$
MONTRED	$(Pc, Ps) = (Pc, Ps) \cdot 2^{-n} \bmod N$
MOD rd,rs0,rs1,rs2	rd = (rs1$\cdot 2^n$ + rs0) mod N, $2^{2n} \bmod N$ initially stored in rs2
MOD_MULT rd,rs0,rs1,rs2	rd = (rs0$\cdot$rs1) mod N, $2^{2n} \bmod N$ initially stored in rs2
MOD_INV rd,rs0	rd = (1/rs0) mod N
MOD_EXP rd,rs0,rs2,length	rd = rs0^{Exp} mod N, Exp has length+1 bits, $2^{2n} \bmod N$ stored in rs2
GF_ADD rd,rs0,rs1	rd = rs0$\oplus$rs1
GF_MULT	$Pc = A \cdot B$
GF_INV	$A = 1/Pc$
GF_INVMULT	$A = B/Pc$
GF_EXP rd,rs0,length	rd = rs0^{Exp} mod N Exp has length+1 bits, $2^{2n} \bmod N$ stored in rs2
EC_ADD rd,rs0,rs1,rs2,wb	(rd,rd+1) = (rs0,rs0+1) + (rs1,rs1+1), curve defined by (R_6, N) NOTE: if (wb = 0) addition is performed but result is discarded
EC_DOUBLE rd,rs0,rs2	(rd,rd+1) = 2$\cdot$(rs0,rs0+1), curve defined by (R_6, N)
EC_MULT length	$(R_4, R_5) = Exp \cdot (R_2, R_3)$, Exp has length+1 bits, curve defined by (R_6, N)

is required as various instructions within the DSRCP's ISA are implemented using other instructions within the ISA, as illustrated for the MOD_MULT instruction in Figure 3. The first tier of control corresponds to those instructions that are implemented directly in hardware. The second tier of control represents the first level of microcode encoded instructions, which are composed of sequences of first tier instructions. Similarly, the third tier of control represents the second level of microcode encoded instructions which consist of sequences of both first and second tier instructions.

The microcode approach is chosen due to its simplicity and extensibility as modifications and enhancements of the ISA can be accomplished with a minimal amount of design effort by modifying the microcode ROM contents. The drawback of using this approach is the additional latency that is incurred by accessing the ROMs, which can end up consuming a significant portion of the processor's cycle time. This performance issue is addressed by pipelining the instruction decoding/sequencing at the output of the first-level microcode ROM.

The global controller is also responsible for disabling unused portions of the circuitry in order to eliminate any unnecessary switched capacitance. The shutdown strategy is dictated by the current width of the processor and enables the datapath to be shutdown in 32 32-bit increments. SET_LENGTH(length), $7 \leq$ length ≤ 1023, is used to set the current width of the processor. All operands

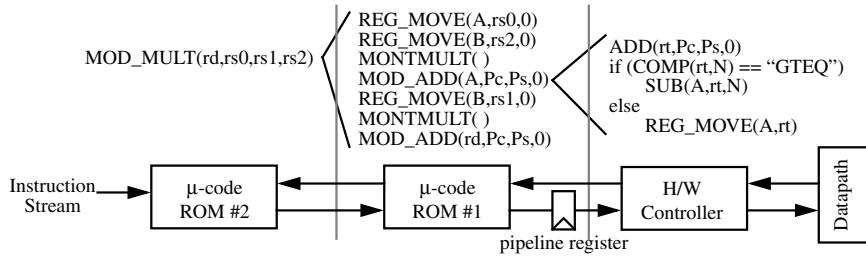

Fig. 3. Hierarchical instruction structure of the DSRCP.

accessed and operated upon by the datapath are assumed to be the size of the current width of the processor, as set by the last invocation of `SET_LENGTH`. This length is then used by the control logic to determine the number of iterations that need to be performed by the various operations.

4.2 I/O Interface

Operands used within the processor can vary in size from 8-1024 bits, requiring the use of a flexible I/O interface that allows the user to transfer data to/from the processor in a very efficient manner. A 32-bit interface is used which is very well suited to existing processors and systems which are predominantly built upon 32-bit interfaces. The choice enables fast operand transfer onto and off of the processor, requiring at most 32 cycles to transfer the largest possible operand.

4.3 Reconfigurable Datapath

The primary component of the DSRCP is the reconfigurable datapath, whose architecture is shown in Figure 4. The datapath consists of four major functional blocks: an eight word register file, a fast adder unit, a comparator unit, and the main reconfigurable computation unit. The datapath is implemented using a very area-efficient bitsliced implementation in order to minimize its size, and the corresponding wiring capacitance of its control signal generation/distribution.

The register file size is chosen to be eight words as it is the minimum number required to implement all of the functions of the datapath. The limiting case for this architecture is that of elliptic curve point multiplication in which registers (R_2,R_3) are used to store the point that is going to be multiplied by the value stored in Exp register, (R_4,R_5) are used to store the result, (R_0,R_1) are used to store an intermediate point used during the computation, R_6 is used to store the curve parameter a, and R_7 is used as a dummy register in order to provide resilience to timing attacks.

The number of read and write ports within the register file is dictated by the requirement to be able to perform single cycle, two operand instructions which generate a writeback value. In certain cases two write ports could have proved

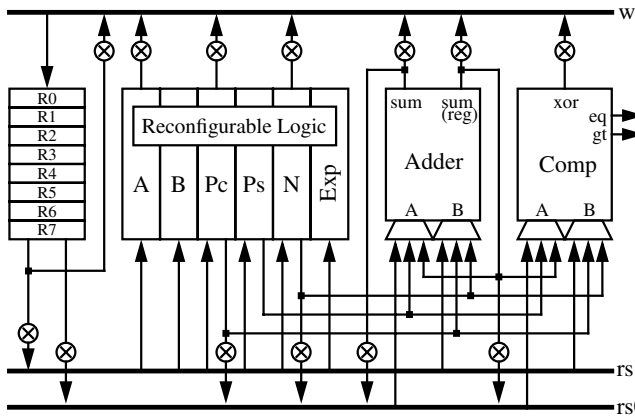

Fig. 4. Reconfigurable datapath architecture block diagram.

useful (e.g., elliptic curve point transfers), but the infrequency of the operation didn't merit the additional overhead that it would have introduced. The register file provides access to the LSB's of R_0, R_1, R_2, and R_3, as required by the modular inversion operation.

The fast adder unit is capable of adding/subtracting two n-bit ($8 \leq n \leq 1024$) operands in four cycles using the hybrid carry-bypass and carry-select technique described in [7] (Figure 5), and optimized for a bitsliced implementation. The unit features a local register to store the previous sum result, a feature that is used in modular addition/subtraction and inversion routines. The adder unit can also right shift its result, as required by the modular inversion algorithm used within the DSRCP.

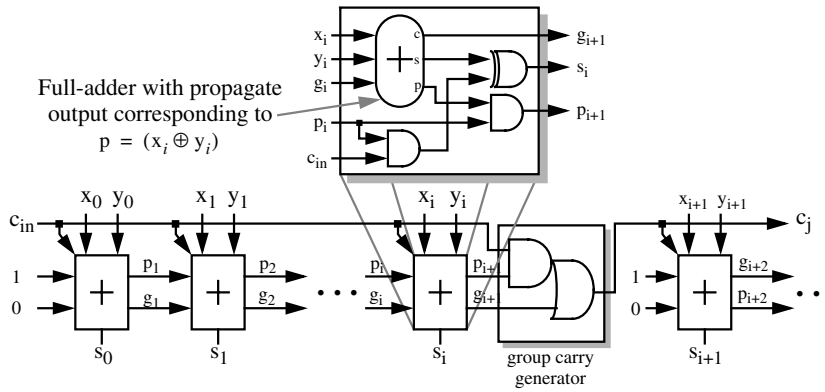

Fig. 5. Modified bitsliced carry-bypass/select adder [7].

The comparator unit performs single-cycle magnitude comparisons between two n-bit operands, as well as computing the XOR of the two operands (i.e., $GF(2^n)$ addition). The comparator generates two flags, `gt` and `eq`, that can be decoded into all possible magnitude relations using a fast $\mathcal{O}(\log_2 n)$ depth tree-based topology that enables single-cycle magnitude comparisons.

The reconfigurable computation unit consists of six local registers (Pc, Ps, A, B, Exp, and N) and a reconfigurable logic block that is capable of implementing all of the required datapath operations. Using local memory within the datapath eliminates the need to continually access the register file every cycle, eliminating the associated overhead of repeated register file accesses. The Pc and Ps registers are used primarily in modular operations to store the carry-save format partial product, and in Galois Field operations as two separate temporary values. A and B store the input operands used in all modular and Galois Field operations. The Exp register is used for storing either the exponent value, in the case of exponentiation operations, or the multiplier value, in the case of elliptic curve point multiplication. The N register also serves a dual purpose; for modular operations it's used as the modulus value, and in Galois Field operations it stores the field polynomial in a binary vector form (e.g., $x^3 + x^2 + 1$ is stored as [1,1,0,1]). In all relevant operations, it's assumed that both the Exp and N registers are pre-loaded with their required values.

5 Reconfigurable Logic Cell Design

The DSRCP is capable of performing a variety of algorithms using both conventional and modular integer fields, as well as binary Galois Fields. These operations are implemented using a single computation unit that can be reconfigured on the fly to perform the required operation. The possible configurations are Montgomery multiplication/reduction, $GF(2^n)$ multiplication, and $GF(2^n)$ inversion. All other operations are either handled by other units such as the fast-adder and comparator, or implemented in microcode.

5.1 Montgomery Multiplication

Montgomery multiplication [8] utilizes the simple iterated radix-2 implementation:

$$(Pc, Ps)_{j+1} = \frac{(Pc, Ps)_j + b_j A + q_j N}{2}, j = 0, \ldots, n-1 \qquad (1)$$

where $q_j = Ps_0 \oplus b_j A_0$, and b_j is the jth bit of operand B. A redundant carry-save representation of the partial product accumulator (Pc,Ps) is exploited in order to minimize the cycle time. This operation can be implemented using the basic computational resources of Figure 6(a): two full-adders and two `AND` gates. Montgomery reduction of A can be performed by setting $B = 1$ (i.e., $b_0 = 1, b_i = 0, i = 1, ..., n-1$). Similarly, reduction of (Pc,Ps) can be performed by setting $B = 0$.

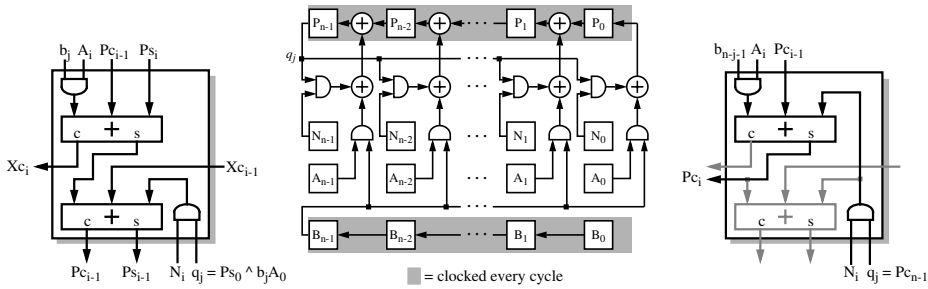

Fig. 6. Multiplication architectures and datapath configurations.

5.2 $GF(2^n)$ Multiplication

Mastrovito's thesis [9] serves as an extensive reference of hardware architectures for performing $GF(2^n)$ multiplication. Given our choice of a polynomial basis, the most efficient multiplier architecture is an MSB-first approach (Figure 6(b) [10]) as it minimizes the number of registers that are clocked in any given cycle. In addition, the MSB-first approach can be mapped to the existing hardware of the Montgomery multiplier (Figure 6(c)) by exploiting the fact that a full-adder's sum output computes a 3-input $GF(2)$ addition. Hence, $GF(2^n)$ multiplication can be performed using the iteration:

$$Pc_j = 2Pc_{j-1} \oplus b_{n-j-1}A \oplus q_jN, j = 0, \ldots, n-1 \tag{2}$$

where q_j is bit $n-1$ of Pc_{j-1}, which is used to modularly reduce the partial product Pc_j. The field polynomial, $f(x)$, is stored as a binary vector in N, and the resulting approach is universal in the sense that it can operate with any valid field polynomial over $GF(2^n)$ for $8 \leq n \leq 1024$.

5.3 $GF(2^n)$ Inversion

The limiting operation in affine co-ordinate Elliptic Curve point operations is typically the inversion operation. In hardware using a polynomial basis, the Extended Binary Euclidean Algorithm [11] can be used to compute inverses in a very efficient manner. This algorithm can also be modified to perform a multiplication in concurrency with the inversion by initializing the Y variable to be the multiplier value (if no multiplication is required, the Y register can simply be initialized with the value 1). This optimization provides significant savings during elliptic curve point operations as it eliminates one multiplication, reducing the total cycle count by approximately 18%. The resulting algorithm (Algorithm 1) can be further optimized by parallelizing the two embedded `while` loops, which effectively halves the number of cycles required as the dominant portion of time is spent in this part of the algorithm. The net result of these

optimizations is a universal invert-and-multiply operation that takes at most four multiplication times ($T_{mult} = n$ cycles), and on average $3.3 \cdot T_{mult}$ in order to invert (and multiply) an element of $GF(2^n)$.

Input: W: a, the element of $GF(2^n)$ that is to be inverted
X: N, the binary representation of the field polynomial $f(x)$
Y: b, the element of $GF(2^n)$ that is to be multiplied by the computed inverse
Z: 0, just plain old zero!

Output: $Z = b/a$

Algorithm:

```
while (W != 0)
   while (W0 == 0)
      W = W/2
      Y = (Y + Y0·N)/2
   endwhile
   while (X0 == 0)
      X = X/2
      Z = (Z + Z0·N)/2
   endwhile
   if (W >= X)
      W = W + X
      Y = Y + Z
   else
      X = W + X
      Z = Y + Z
   endif
endwhile
```

Algorithm 1. Extended Binary Euclidean Algorithm over $GF(2^n)$ used in DSRCP.

Inversion can be implemented with the datapath cell used in both Montgomery and $GF(2^n)$ multiplication by providing a small degree of reconfigurability such that computational resources can be re-used to perform different parts of Algorithm 1. The basic requirements are two 2-input adders over $GF(2)$ to perform each of the parallel `while` loops, and the two summations in each branch of the if clause. Each iteration of the parallel `while` loops requires one cycle for performing the actual operations as all operations are performed in parallel. An additional cycle is incurred when the exit condition of the parallel `while` loops is satisfied (i.e., $W_0 = X_0 = 1$) as it must be detected via an additional iteration of the loop. The second part of the algorithm requires a single cycle as well. The two datapath adders can be used as two-input $GF(2)$ adders by zeroing one of their inputs, and then utilizing multiplexors to allow the adder inputs to be changed on the fly to accommodate Algorithm 1. The corresponding architecture and its resulting mapping to the datapath cell is shown in Figure 7.

The final datapath cell is shown in Figure 8. In all it contains two full-adders, two `AND` gates, 6 two input multiplexors, and 6 register cells. The reconfiguration muxes are controlled through the use of 8 control lines (three for the adder muxes and five for the register muxes).

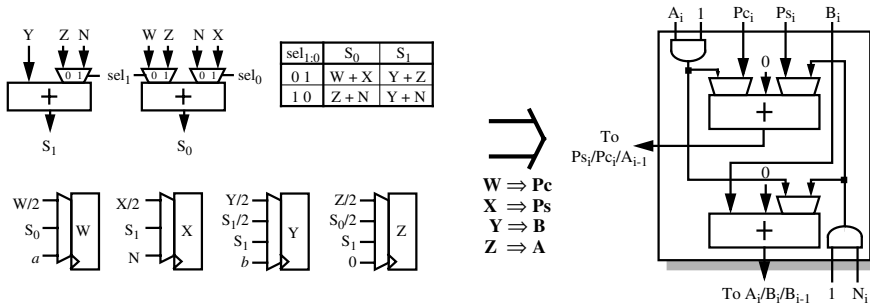

Fig. 7. Basic $GF(2^n)$ inversion architecture and resulting datapath cell.

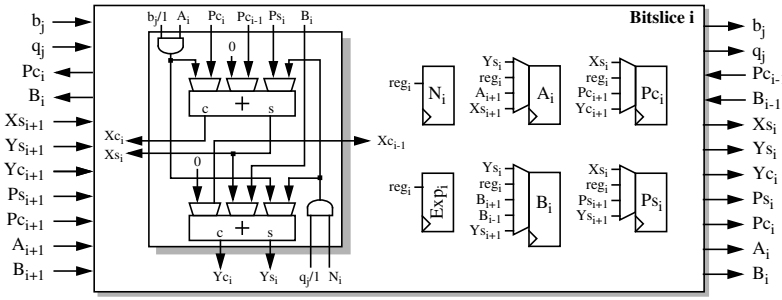

Fig. 8. Final reconfigurable datapath cell.

6 Algorithm Implementation

The DSRCP performs a variety of algorithms ranging from modular integer arithmetic to Elliptic Curve arithmetic over $GF(2^n)$. All operations are universal in that they can be performed using any valid n-bit modulus or $GF(2^n)$ field polynomial, with $8 \leq n \leq 1024$.

6.1 Modular Arithmetic

The various complex modular arithmetic operations (multiplication, reduction, inversion, and exponentiation) are implemented using microcode. Multiplication is performed using Montgomery multiplication, which requires a correction factor of 2^{2n} mod N be provided with the modulus N in order to undo the division by 2^n inherent in Montgomery's method.

Modular reduction is performed using Montgomery reduction and exploits the fact that the value being reduced can be decomposed into two n-bit values and then reduced via Algorithm 2.

Modular inverses are computed using the Extended Binary Euclidean Algorithm. This technique requires special architectural considerations such as the

Input: rs0, rs1: n-bit registers containing the value to be reduced stored in the format $(\text{rs1}\cdot 2^n + \text{rs0})$
rs2: n-bit register containing the Montgomery correction value $2^{2n} \bmod N$

Output: rd = $(\text{rs1}\cdot 2^n + \text{rs0}) \bmod N$

```
Algorithm: REG_MOVE(Ps,rs0), CLEAR_PC  // Ps = rs0, Pc = 0
           MONTRED( )                  // (Pc,Ps) = rs0·2^-n mod N
           MOD_ADD(rd, Pc,Ps,0)        // rd = rs0·2^-n mod N
           MOD_ADD(rd,rd,rs1,0)        // rd = (rs1·2^n + rs0)2^-n mod N
           REG_MOVE(A,rd)              // A = rd
           REG_MOVE(B,rs2)             // B = 2^2n mod N
           MONTMULT( )                 // (Pc,Ps) = (rs1·2^n + rs0) mod N
           MOD_ADD(rd,Pc,Ps,0)         // rd = (rs1·2^n + rs0) mod N
```

Algorithm 2. Modular reduction implementation on the DSRCP.

ability to right shift the output of the adder unit, and explicit access to the LSB of R_0, R_1, R_2, and R_3 in order to check the looping conditions of the Euclidean algorithm.

Modular exponentiation is performed using a standard square-and-multiply algorithm [12] with an exponent scanning window of size two. The algorithm (Algorithm 3) pre-computes and stores the values $\{2^n, rs0 \cdot 2^n, rs0^2 \cdot 2^n, rs0^3 \cdot 2^n\}$ in $\{R_0, R_1, R_2, R_3\}$ respectively. During each iteration the current value is squared twice, and then the exponent is scanned two bits at a time (this scanning is done non-destructively so exponent values don't need to be reloaded prior to each operation). The value scanned corresponds to the register that is used during the subsequent multiplication (e.g., if `01` is read then R_1 is used). Note that multiplying by the value stored in R_0 is essentially a NOP as Montgomery multiplication is being used which implicitly divides the product by 2^n.

The use of NOPs provides protection from timing attacks and simple power analysis as a multiplication is always performed, thereby eliminating any variation in execution based on the exponent's value. The expense of this immunity is that strings of zeros in the exponent cannot be exploited to speed up the operation. The loss in efficiency due to this fixed performance, assuming that the exponent is uniformly distributed, is only 9%.

The use of the length operand in the `MOD_EXP` instruction enables the length of the exponent and the operands to be decoupled, leading to much more efficient exponentiation when the exponent value is significantly shorter than the operands.

6.2 $GF(2^n)$ Arithmetic

$GF(2^n)$ addition is performed using the XOR function of the comparator unit, and both $GF(2^n)$ multiplication and inversion are implemented directly in hardware using the reconfigurable datapath. $GF(2^n)$ exponentiation is implemented in the same manner as modular exponentiation, with $\{1, rs0, rs0^2, rs0^3\}$ being pre-computed and stored in $\{R_0, R_1, R_2, R_3\}$. NOPs are once again exploited to provide immunity to timing attacks and simple power analysis.

Input: rs0: n-bit register containing value to be exponentiated
rs2: n-bit register containing Montgomery correction value $2^{2n} \bmod N$
length: 10-bit value representing length of the exponent stored in Exp

Output: rd = rs0$^{Exp} \bmod N$

Algorithm:

```
REG_MOVE(Ps,rs2)                    // Ps = 2^2n mod N
MONTRED( )                          // (Pc,Ps) = 2^n mod N
MOD_ADD(R0,Pc,Ps,0)                 // R0 = 2^n mod N
REG_MOVE(A,rs0)                     // A = rs0
REG_MOVE(B,rs2)                     // B = 2^2n mod N
MONTMULT( )                         // (Pc,Ps) = rs0·2^n mod N
MOD_ADD(R1,Pc,Ps,0)                 // R1 = rs0·2^n mod N
REG_MOVE(A/B,R1)                    // A,B = rs0·2^n mod N
MONTMULT( )                         // (Pc,Ps) = rs0^2·2^n mod N
MOD_ADD(R2,Pc,Ps,0)                 // R2 = rs0^2·2^n mod N
REG_MOVE(B,R2)                      // B = rs0^2·2^n mod N
MONTMULT( )                         // (Pc,Ps) = rs0^3·2^n mod N
MOD_ADD(R3,Pc,Ps,0)                 // R3 = rs0^3·2^n mod N
REG_MOVE(A/B,R0)                    // A,B = 2^n mod N
for (i = length-1; i >= 0;i = i-2)
   MONTMULT( )                      // (Pc,Ps) = P^2·2^n mod N
   MOD_ADD(A/B,Pc,Ps,0)             // A,B = P^2·2^n mod N
   MONTMULT( )                      // (Pc,Ps) = P^4·2^n mod N
   MOD_ADD(A,Pc,Ps,0)               // A = P^4·2^n mod N
   REG_MOVE(B,R<Exp2i:2i−1>)        // B = R<Exp2i:2i−1> = Rj
   MONTMULT( )                      // (Pc,Ps) = P^4+j·2^n mod N
   MOD_ADD(A/B,Pc,Ps,0)             // A,B = P^4+j·2^n mod N
endfor
MONTRED_A( )                      // (Pc,Ps) = P^Exp mod N
MOD_ADD(rd,Pc,Ps,0)                 // rd = P^Exp mod N
```

Algorithm 3. Modular exponentiation on the DSRCP.

6.3 Elliptic Curve

The DSRCP performs affine coordinate elliptic curve operations on non-supersingular curves over $GF(2^n)$ of the form:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{3}$$

where $a, b \in GF(2^n)$. The corresponding point addition and doubling formulae, assuming that P_1 and P_2 are distinct points on E, are given by:

$$\begin{aligned} P_1 + P_2 = (x_3, y_3), x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_2 + x_3)\lambda + x_3 + y_2 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned} \tag{4}$$

$$\begin{aligned} 2P_1 = (x_3, y_3), x_3 &= \lambda^2 + \lambda + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{5}$$

Note that the ISA of the DSRCP enables it to also perform elliptic curve operations over fields of prime characteristic using an external sequencer and the appropriate formulae (e.g., [13]).

Point addition and doubling are implemented in microcode using the above formulae, with curve points stored as register pairs $(R_i, R_{i+1}) = (x, y)$. Point addition features an additional input in the form of a writeback enable bit which must be set for the result to be written back to the destination register pair. If the enable bit is not set, then the computation is performed and the result is discarded, leaving the destination register pair unaffected. This feature is used to provide immunity to timing attacks and simple power analysis during elliptic curve point multiplication.

Point multiplication is performed using a repeated double-and-add algorithm, with a window size of one. Larger window sizes are not possible on the current DSRCP architecture due to memory limitations of the register file (e.g., four pre-computed values would require 8 additional registers). The issue of timing attacks is once again addressed by utilizing NOPs via the writeback enable bit of the point addition operation. The overhead associated with using NOPs is 33% relative to a conventional implementation where NOPs are skipped, and 50% if a signed radix-2 representation is used for the multiplier [12].

7 Performance Estimates

Cycle counts for the proposed architecture are determined via simulation using Verilog and the Synopsys Timemill^{TM} simulator [14]. The results are shown in Figure 9 for the various operations in terms of the cycles per bit of the operand (e.g., 1024-bit modular multiplication takes approximately 2048 cycles). The execution times for the modular/$GF(2^n)$ exponentiation and elliptic curve multiplication operations are derived using a nominal operating frequency of 50 MHz, at a supply voltage of 1 V. The power consumption of the datapath has also been simulated using Synopsys' Powermill^{TM} simulator [15], and is estimated to be at most 15 mW using the aforementioned operating conditions.

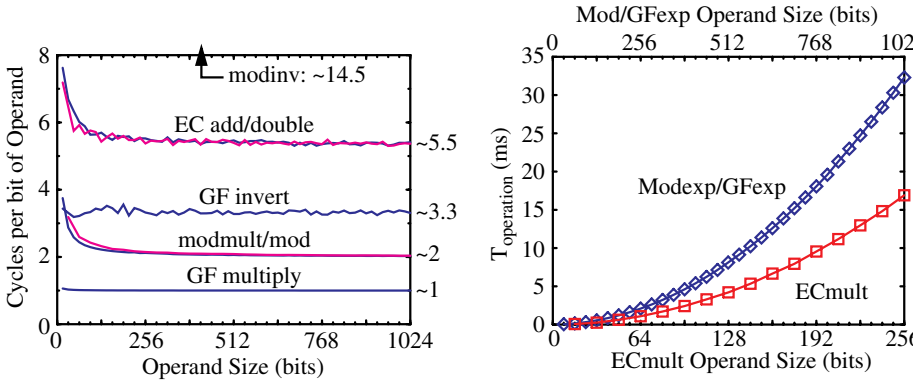

Fig. 9. Simulated performance of the DSRCP.

The SHA-1 hash function engine's performance under these conditions is $266 Mbps$ at a peak power consumption of 800 μW, or 3 pJ/bit. In comparison, an optimized assembly language software-based solution executing on Intel's StrongARM^{TM} SA-1100 [16] has a rate of 33.9 $Mbps$ at a power consumption of 352.5 mW, for 10.4 nJ/bit. Hence, the hardware-based solution described here is over three orders of magnitude more energy efficient.

8 Results and Conclusions

The resulting estimated energy consumption of the DSRCP for a variety of operations and operand sizes is presented in Figure 10. Energy estimates for conventional FPGA and software based solutions are also depicted for comparison. The energy estimates for the FPGA's come from estimates based on the work described in [18] and [17]. The software-based energy consumptions were measured using a StrongARM^{TM} SA-1100 evaluation platform that was executing hand-optimized assembly language implementations of the various operations.

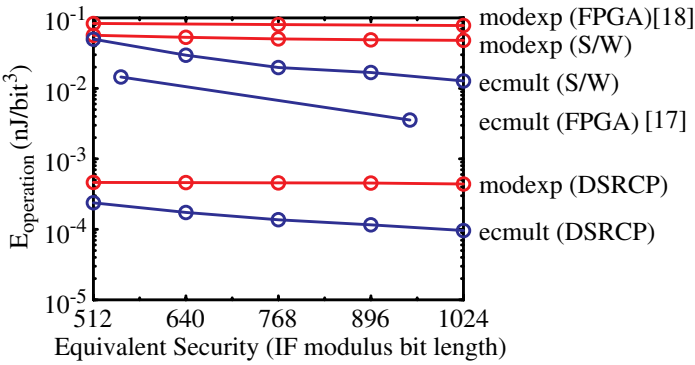

Fig. 10. Estimated energy efficiency of the DSRCP vs. conventional solutions (FPGA & S/W).

The comparison illustrates that the DSRCP is estimated to be on the order of $30-180\times$ more energy efficient than generic FPGA-based solutions, and over two orders of magnitude more energy efficient than conventional software-based solutions. In addition, the proposed architecture enables the user the same flexibility as both the software and FPGA-based solutions in terms of implementing asymmetric cryptographic algorithms.

Acknowledgements

Effort sponsored by National Semiconductor, as well as the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air

Force Materiel Command, USAF, under agreement number F30602-00-2-0551. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

1. IEEE P1363, "Standard specifications for public-key cryptography - Draft 13," IEEE, November 12, 1999.
2. Xilinx Corporation, *Virtex-E Field Programmable Gate Arrays (XCV00) Databook*, 1999.
3. Altera Corporation, *APEX 20K Programmable Logic Device Family Data Sheet*, 2000.
4. E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," *ISLPED'98 - Proceedings of the 1998 International Symposium on Low Power Electronic Design*, 1998, 155–160.
5. Xilinx Corporation, *XC4000 Field Programmable Gate Arrays: Programmable Logic Databook*, 1996.
6. FIPS 180-1, "Secure hash standard," Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia, April 17, 1995.
7. A. Satoh, *et. al.*, "A high-speed small RSA encryption LSI with low power dissipation," *ISW'97 - Proceedings of First International Information Security Workshop*, 1998, 174–187.
8. P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, 48 (1987), 243–264
9. E.D. Mastrovito, *VLSI Architectures for Computation in Galois Fields*, Ph.D. Thesis, Linköping University, Linköping, Sweden, 1991.
10. P.A. Scott, S.E. Tavares, and L.E. Peppard, "A fast VLSI multiplier for $GF(2^m)$," *IEEE Journal on Selected Areas of Communications*, vol. SAC-4, no.1, January 1986, 62–66.
11. D.E. Knuth, *The Art of Computer Programming - Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading MA, 2nd Edition, 1981.
12. D.M. Gordon, "A survey of fast exponentiation methods," *Journal of Algorithms*, vol. 27, no. 1, April 1998, 129–146.
13. G. Seroussi, N.P. Smart, and I.F. Blake, *Elliptic Curve Cryptography*, Cambridge University Press, February 2000.
14. Synopys Corporation, *TimeMill User's Manual*, 1999.
15. Synopsys Corporation, *PowerMill User's Manual*, 1999.
16. Intel Corporation, *StrongARM SA-1100 Microprocessor for Portable Applications Brief Datasheet*, September 1999.
17. M. Rosner, *Elliptic Curve Cryptosystems on Reconfigurable Hardware*, Master's Thesis, Worcester Polytechnic Institute, Worcester MA, 1998.
18. T. Blum, *Modular Exponentiation on Reconfigurable Hardware*, Master's Thesis, Worcester Polytechnic Institute, Worcester MA, 1999.