Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 390176, 8 pages
http://dx.doi.org/10.1155/2014/390176

Research Article

System Level Design of Reconfigurable Server Farms Using Elliptic Curve Cryptography Processor Engines

Sangook Moon[1] and Jongsu Park[2]

[1] *Department of Electronic Engineering, Mokwon University, 88 Doanbuk-ro, Seo-gu, Daejeon 302-729, Republic of Korea*
[2] *School of Electrical and Electronic Engineering, Yonsei University, Seoul 120-749, Republic of Korea*

Correspondence should be addressed to Sangook Moon; smoon@mokwon.ac.kr

Received 15 November 2013; Accepted 15 April 2014; Published 5 May 2014

Academic Editor: Jongsung Kim

Copyright © 2014 S. Moon and J. Park. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

As today's hardware architecture becomes more and more complicated, it is getting harder to modify or improve the microarchitecture of a design in register transfer level (RTL). Consequently, traditional methods we have used to develop a design are not capable of coping with complex designs. In this paper, we suggest a way of designing complex digital logic circuits with a soft and advanced type of SystemVerilog at an electronic system level. We apply the concept of design-and-reuse with a high level of abstraction to implement elliptic curve crypto-processor server farms. With the concept of the superior level of abstraction to the RTL used with the traditional HDL design, we successfully achieved the soft implementation of the crypto-processor server farms as well as robust test bench code with trivial effort in the same simulation environment. Otherwise, it could have required error-prone Verilog simulations for the hardware IPs and other time-consuming jobs such as C/SystemC verification for the software, sacrificing more time and effort. In the design of the elliptic curve cryptography processor engine, we propose a 3X faster GF(2^m) serial multiplication architecture.

1. Introduction

Electronic system level (ESL) design includes hardware and software interactions with higher levels of abstraction for system-level transactions. ESL methodologies have been evolved from algorithmic modeling such as architectural explorations and proving concepts as supplementary techniques such as design of embedded systems, system-level test bench development, hardware/software cosimulation, and high-level synthesis of ASIC/FPGA designs. Efficient ESL design includes the ability to proceed from the concept to the optimal implementation of architectural functionality, as well as verification. As one of the most evolved ESL languages, Bluespec SystemVerilog (BSV) has introduced to the system hardware architects a new way to simplify implementing the complicated control logic while maintaining control over the architecture and efficiency of the design at the same time. According to [1], over 50% of reduction in time can be achieved to verify a design and less than 50% of the bugs can be found compared to traditional RTL design. BSV differs from traditional Verilog or VHDL, or even from SystemVerilog in many aspects. Instead of using the traditional procedural statements which is to implementhardware concurrency such as *always*, BSV uses statements named *rules* that mean atomic behaviors which are fully synthesizable. BSV enables a good overall system generation since all instances such as *methods, rules, modules, interfaces* and *functions* are regarded as the first-class objects. This means that an object can be used as an argument to another object.

In this work, we adopt a previous work of our elliptic curve crypto-processor architecture [2], in which we propose a 3X faster finite field multiplier. In order to reuse the functional unit architecture, we implemented a wrapper which enabled the functional unit to be used in BSV and extracted a higher level of abstraction. Building a server farm in the traditional way requires a complex hardware finite state machine in HDL, which costs time and effort. By using a high-level abstraction language such as BSV, however, it is far easier to define the state machine logic and explore the design. We experimentally implemented crypto-server farms using BSV with existing Verilog IP logics such as an elliptic

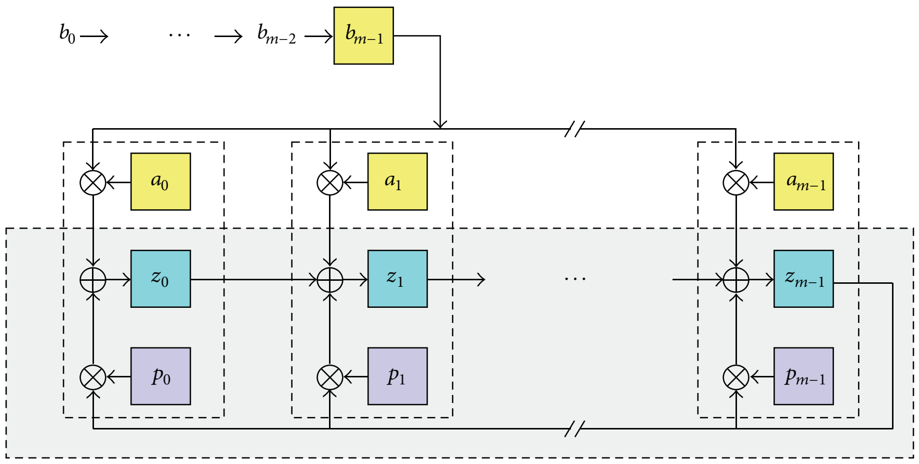

FIGURE 1: Basic structure of Mastrovito's serial multiplication.

curve cryptoprocessor, a random number generator, to build the elliptic curve crypto-processor server farms. We firstly discuss the design of the elliptic curve crypto-processor and then we discuss how to utilize and apply the IPs in BSV test bench program. Next, after considering how we achieved the higher level of abstraction, we present the random number generator, the reorder buffer, and the architecture of the server farms and finally discuss the result.

2. 193-Bit Elliptic Curve Cryptoengine

2.1. 3X Faster Finite Field Multiplication. The method is based on the Mastrovito's serial GF (Galois field) multiplier architecture [3]. We represent the elements of the GF(2^m) as polynomial basis. Suppose two elements of given GF(2^m) are represented as $A(x) = \sum_{i=0}^{m-1} a_i x^i$ and $B(x) = \sum_{i=0}^{m-1} b_i x^i$, respectively. Then the GF multiplication can be represented as (1). Figure 1 shows the basic structure of Mastrovito's serial GF multiplier:

$$\begin{aligned} Z(x) &= A(x)\sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} b_i \left(x^i A(x)\right) \bmod P(x) \\ &= \left[b_0 A(x) + \cdots + b_{m-2}x^{m-2}A(x) \right. \\ &\quad \left. + b_{m-1}x^{m-1}A(x)\right] \bmod P(x). \end{aligned} \tag{1}$$

Now, we can derive a group of expressions (2) for 3X faster GF multiplier as follows:

$$\begin{aligned} &Z_{3k}(x) \\ &\quad = \left[b_0 A(x) + b_3 x^3 A(x) + b_6 x^6 A(x) + \cdots\right] \bmod P(x) \\ &Z_{3k+1}(x) \\ &\quad = x\left[b_1 A(x) + b_4 x^3 A(x) + b_7 x^6 A(x) + \cdots\right] \bmod P(x) \\ &Z_{3k+2}(x) \\ &\quad = x^2\left[b_2 A(x) + b_5 x^3 A(x) + b_8 x^6 A(x) + \cdots\right] \bmod P(x). \end{aligned} \tag{2}$$

Analyzing the first equation of expression (2), the structure follows the basic scheme of Mastrovito's serial multiplication, except that the orders of x are multiples of 3. The second and the third equations of (2) are similar to the first, except that the bases of the coefficients b are different and postmultiplication of x and x^2 should be processed.

To realize the implementation of 3X faster GF multiplication, we started to derive an equation for calculating $x^3A(x)$ from expression (3). After that, we induced equations for x^m, x^{m+1}, and x^{m+2} as shown in expression (4). Consider

$$\begin{aligned} x^3 A(x) &= a_0 x^3 + a_1 x^4 + \cdots + a_{m-4}x^{m-1} + a_{m-3}x^m \\ &\quad + a_{m-2}x^{m+1} + a_{m-1}x^{m+2} \end{aligned} \tag{3}$$

$$\begin{aligned} x^m &= p_0 + p_1 x + \cdots + p_{m-2}x^{m-2} + p_{m-1}x^{m-1} \\ x^{m+1} &= p_0 x + p_1 x^2 + \cdots + p_{m-2}x^{m-1} + p_{m-1}x^m \\ x^{m+2} &= p_0 x^2 + p_1 x^3 + \cdots + p_{m-2}x^m + p_{m-1}x^{m+1}. \end{aligned} \tag{4}$$

Here, without losing generality, we can use trinomials or pentanomials as the primitive polynomial in this finite field [4], where both p_{m-1} and p_{m-2} are zero. Thus, applying reduced version of (4) to (3), $x^3A(x)$ can be derived as expression (5):

$$\begin{aligned} x^3 A(x) &= a_{m-3}p_0 + \left(a_{m-3}p_1 + a_{m-2}p_0\right)x \\ &\quad + \left(a_{m-3}p_2 + a_{m-2}p_1 + a_{m-1}p_0\right)x^2 \\ &\quad + \left(a_0 + a_{m-3}p_3 + a_{m-2}p_2 + a_{m-1}p_1\right)x^3 \\ &\quad + \left(a_1 + a_{m-3}p_4 + a_{m-2}p_3 + a_{m-1}p_2\right)x^4 \\ &\quad + \cdots + \left(a_{m-4} + a_{m-3}p_{m-1} \right. \\ &\quad \left. + a_{m-2}p_{m-2} + a_{m-1}p_{m-3}\right)x^{m-1}. \end{aligned} \tag{5}$$

Expression (5) can be represented as circuits. Figure 2 shows the x^3-multiplying circuit which plays a key role in 3X faster GF multiplier.

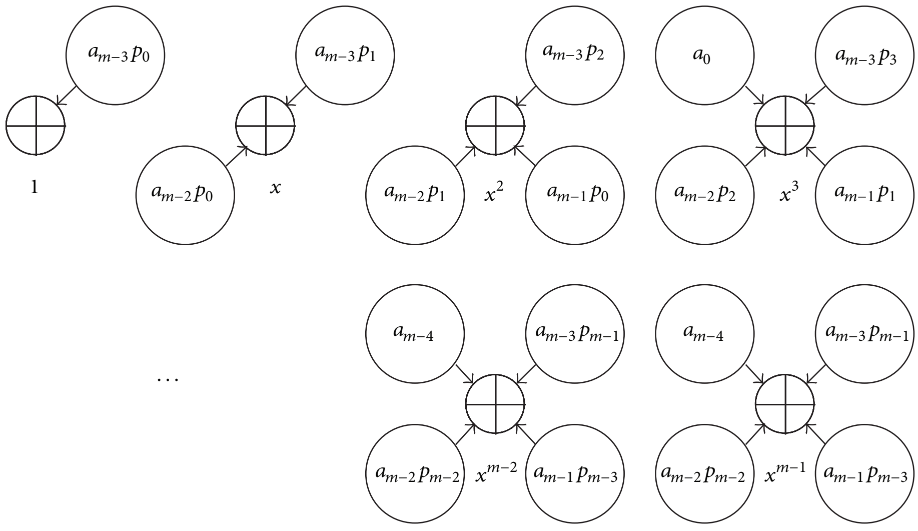

FIGURE 2: Proposed x^3-multiplying circuit.

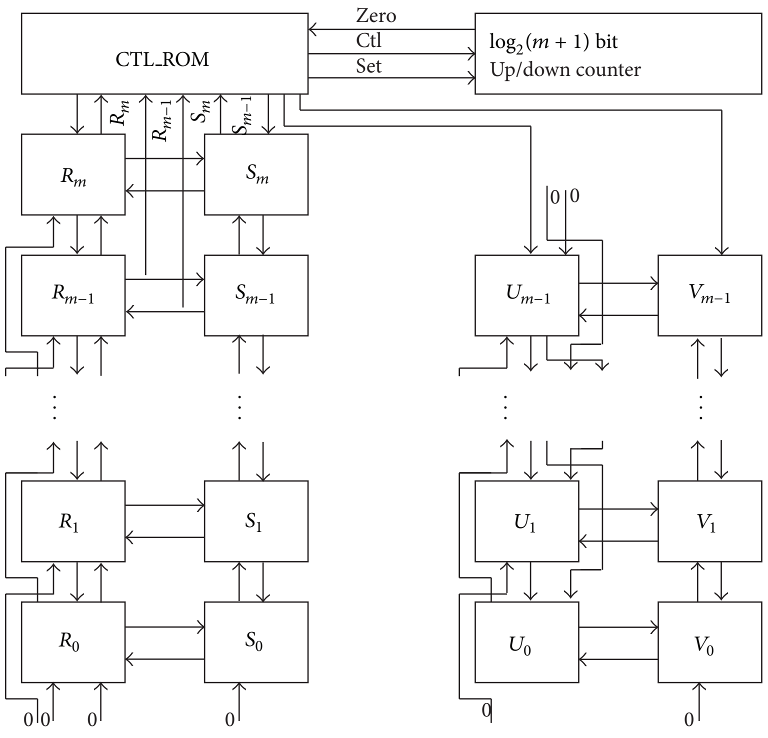

FIGURE 3: Block diagram of GF(2^m) divider with half-iteration.

2.2. Finite Field Division. Division in GF(2^m) of $A(x)/B(x)$ can be represented as $A(x)$ times the inverse of $B(x)$ ($A(x) \cdot B(x)^{-1}$). There are two methods to find the inverse of an element in GF(2^m). One is table lookup method, and the other is algorithm-based method.

Table lookup method is efficient for small m, but the size of the table is exponentially proportional to m, resulting in gigantic circuit size.

Algorithm-based method is utilizing various mathematical algorithms such as Fermat's theorem and Euclid's theorem. Since Fermat's theorem is effective for only small m [5], we decided to adopt Euclid's and defined the microarchitecture to reduce the number of iterations to half of the original algorithm.

The block diagram of our half-iteration divider is shown in Figure 3. For each R, S, U, and V cell, we designed functional circuits corresponding to 19 possible combinational operations. The divider consists of $m + 1$ number of R and S cells, m number of U and V cells, a counter, and a controller. The combinational operations are listed and summarized in Table 1.

TABLE 1: GF(2^m) divider combinational operations.

Number	R_mR_{m-1}	S_mS_{m-1}	D	R	S	U	V	Delta
1	"00"	xx	X	x^2R	S	x^2U	Equal	Delta + 2
2	"01"	"0x"	X	xR	xS	Equal	Equal	Equal
3	"01"	"1x"	X	xR	$x(S-xR)$	Equal	$V-xU$	Equal
4	"10"	"00"	0	x^2S	R	x^2V	U	2
5	"10"	"00"	1	R	x^2S	U/x^2	Equal	Delta – 2
6	"10"	"01"	0	xS	$x(xS-R)$	V	$U-xV$	0
7	"10"	"01"	1	R	$x(xS-R)$	U/x^2	$V-U/x$	Delta – 2
8	"10"	"10"	0	$x^2(S-R)$	R	$x^2(V-U)$	U	2
9	"10"	"10"	1	R	$x^2(S-R)$	U/x^2	$V-U$	Delta – 2
10	"10"	"11"	0	$x(S-R)$	$x(R-x(S-R))$	$V-U$	$U-x(V-U)$	0
11	"10"	"11"	1	R	$x(R-x(S-R))$	U/x^2	$V-U-U/x$	Delta – 2
12	"11"	"00"	0	x^2S	R	x^2V	U	2
13	"11"	"00"	1	R	x^2S	U/x^2	V	Delta – 2
14	"11"	"01"	0	xS	$x(xS-R)$	V	$U-xV$	0
15	"11"	"01"	1	R	$x(xS-R)$	U/x^2	$V-U/x$	Delta – 2
16	"11"	"10"	0	$x(S-R)$	$x(R-x(S-R))$	$V-U$	$U-x(V-U)$	0
17	"11"	"10"	1	R	$x(R-x(S-R))$	U/x^2	$V-U-U/x$	Delta – 2
18	"11"	"11"	0	$x^2(S-R)$	R	$x^2(V-U)$	U	2
19	"11"	"11"	1	R	$x^2(S-R)$	U/x^2	$V-U$	Delta – 2

The operation of the GF(2^m) divider is as follows.

(1) Input the value of dividend ($A(x)$) and divisor ($B(x)$) into U and R cells, respectively.

(2) Each cell performs its own operation according to the control signals coming from CTL_ROM, depending on 5-bit cell output signals (R_m, R_{m-1}, S_m, S_{m-1}, and zero).

(3) After m cycles, the divider yields the output from U cell.

2.3. 193-Bit Elliptic Curve Cryptography Engine IP. Putting 3X faster GF multiplier and the GF divider with half number of iterations together along with an algorithm for calculating $k{\cdot}P$ (scalar multiplication in elliptic curve cryptography where k is the order of the base point and P is the point to encrypt), we have implemented a 193-bit elliptic curve crypto-processor engine for server farms. Required elliptic curve parameters such as the order of the base point k, the irreducible prime polynomial $p(x) = x^{193} + x^{15} + 1$, and the base point G are given in SEG2 for the 193-bit wide GF(2^{193}) [6].

Figure 4 depicts the 193-bit elliptic curve cryptography processing engine. There are two parts of registers. In parameter registers part, elliptic curve parameters such as public key k, coefficients ($a, b,$ and f), the point to be encrypted (x_p, y_p), and the scalar multiple of two of the points to be encrypted (x_{2p}, y_{2p}) are preloaded to save time. Temporary registers store intermediate results during the scalar multiplication. Add block performs finite field addition and subtraction, with simple XOR gates. 3X faster GF(2^{193}) multiplier produces the GF multiplication result in 65 cycles and 2X faster GF(2^{193}) divider accomplishes the function in 97 cycles.

The purpose of the cryptography engine is to calculate $k \cdot P$. In order to obtain fast calculation results, we applied radix-4 redundant recoding to the binary presentation of elliptic curve point P. Due to the characteristic of radix-4 redundancy recoding, the total number of steps decreases down to $\lceil 193/2 \rceil - 1 = 96$. According to the result of the radix-4 recoding of point P in each step, one out of the Add block outputs $0P$, $\pm P$, $\pm 2P$ is chosen so that we can look up 2 bits simultaneously as is done in the Booth's algorithm [7]. Thus we get the final scalar point multiplication result in 96 clock cycles.

3. Higher Level of Abstraction

3.1. Strong Type System. Languages with higher levels of abstraction such as Haskell suggest a strong "type" system [8]. Every type in every single expression is determined before compiling time, or during static elaboration, which leads to safe code. In such languages, for example, a program in which a Boolean number is divided by an integer fails to be compiled because "Boolean" type differs from "integer" type and try to use the same operator "/", causing type mismatch. This characteristic is recommendable because it is better to catch this kind of error during static elaboration, rather than having a dead lock during execution time. BSV complies with Haskell type system and every keyword belongs to a specific type. So during the static elaboration, the compiler determines such type-check errors, so that we can avoid so many unnecessary error-prone situations.

3.2. Guarded Command Language. BSV also adopts the characteristics of *guarded commanded language* invented by Dijkstra [9]. Since this language is rather a theory than

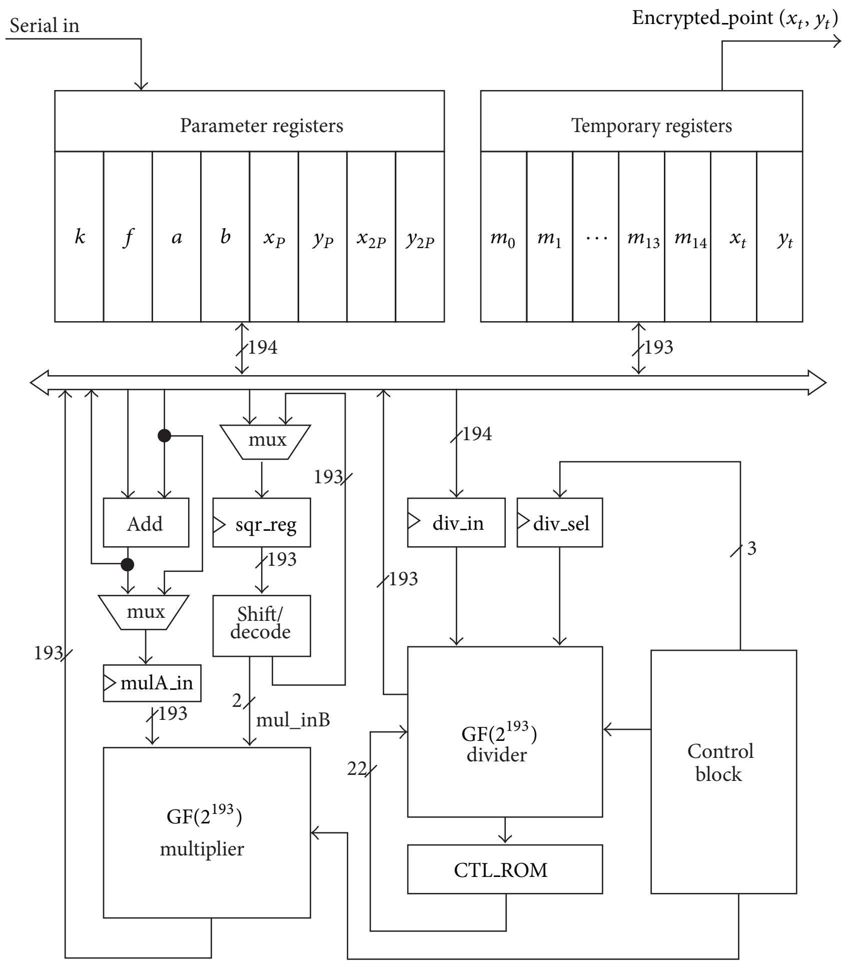

Figure 4: 193-Bit elliptic curve cryptography encryption processor engine.

a physical language, there is no specific compiler. The theory makes the programming concept succinctly integrated, so that we can easily determine the correctness of the program using Hoare logic [10].

Guarded command means that commands used in a program are guarded, as the name implies. Guard is a proposition and it must be true before executing the commands. If the guard is false, the command will not be executed. Guarded command helps prove whether a program satisfies a certain specification or not easily, as shown in the examples in Algorithm 1.

```
Example 1.
if $a < b$ then $c$:= True
Else $c$:= False

is translated as

if $a < b \rightarrow c$:= True
[] $a \geq b \rightarrow c$:= False
Fi

Example 2.
if error = True then $x$:= 0

is translated as

if error = True $\rightarrow$ $x$:= 0
[] error = False $\rightarrow$ skip
fi
```

Algorithm 1: Two examples of guarded command language.

3.3. IP Wrapping to BSV. Hardware circuits described in Verilog use wires and registers to define the input and the output signals of a module. BSV utilizes the object oriented programming's interface-based design to connect between modules. We adopted the concept of *interface* used in BSV that a *method* defines the input and output arguments with strong types and return the value of the *method* as the output with given type. As a result, no more error-prone repetitive port declarations are necessary. Instead, succinct manipulation of interfaces between modules is feasible.

First, we described the 193-bit elliptic curve cryptoengine with Verilog HDL, which is shown in Algorithm 2. In order to extract the highest level of abstraction existing in the state-of-the-art programming languages, we created an appropriate wrapper for the cryptoengine IP to be used in BSV simulator. As is shown in Algorithm 3, we used the BSV feature of

```
module ECC (encrypted_point, ready_ecc, serial_in, encrypt, clock, reset);
output [385:0] encrypted_point;
output ready_ecc

input serial_in, encrypt, clock, reset;
...
endmodule
```

Algorithm 2: Verilog description of the elliptic curve cryptoengine.

```
import GetPut::*;
import ClientServer::*;

interface ECC_Server#(type req_type, type resp_type);
    interface Put#(req_type) request;
    interface Get#(resp_type) response;
endinterface
...

import "BVI" ECC =
module mkECC    (ECC_Server#(req_type, resp_type))
        provisos(Bits#(req_type, s_req), Bits#(resp_type, s_resp));

        default_clock clk_clock;
        ...
        interface Put request;
                method put (serial_in )
                enable(encrypt) clocked_by(clk_clock) reset_by(rst_reset);
        endinterface
        interface Get response;
                method encrypted_point get( ) /* 385:0 */
                  ready(ready_ecc) clocked_by(clk_clock) reset_by(rst_reset);
        endinterface
        schedule request_put CF response_get;
        ...
endmodule
```

Algorithm 3: BSV wrapper including the interfaces, modules, and methods.

"import BVI" mechanism to create the wrapper around the RTL module so that the IP behaves like a BSV module. Instead of using ports, the wrapped module uses *method*s and *interface*s.

4. Implementation

4.1. Elliptic Curve Cryptography Processor Server Farms. A server farm or server cluster is a collection of computer servers with an arbiter to accomplish the server needs far beyond the capability of one machine. The arbiter allocates incoming jobs to any server, which becomes available, and sends the result back to its caller. We implemented two server farms, each of which is equipped with four elliptic curve cryptography processor engines (Figure 5). In the test bench, the arbitration controller sends the same random jobs to each farm and checks to see if corresponding results returned from each crypto-server farm are matched. For both servers, the time required to complete each job depends on the value of the point to be encrypted on the elliptic curve, so we cannot be sure that results will be available in the order the jobs were started. However, the arbiter should return the results to its caller in the order the jobs were received. So a reorder buffer is required, which fits this specification.

4.2. Reorder Buffer. Reorder buffers are required for organizing the order from out-of-order job executions. In BSV, the reorder buffer is implemented in the library package as a reusable parameterized IP [11]. The IP named CompletionBuffer provides the function of reorder buffers. The interface of the IP offers three methods (Algorithm 4). The *reserve* method allows the caller to reserve a slot in the buffer, returning a token holding the identity of the slot.

When a job finishes, the *complete* method allows the result to be stored in the reorder buffer. The *drain* method returns the results in the order where the tokens were assigned from the first. In this way, the results of out-of-orderly completed jobs can wait in the buffer until a time-consuming job ahead of them finishes.

```
interface CompletionBuffer #(numeric type n, type element_type);
        interface Get#(CBToken#(n)) reserve;
        interface Put#(Tuple2 #(CBToken#(n), element_type)) complete;
        interface Get#(element_type) drain;
endinterface: CompletionBuffer
```

Algorithm 4: Interface of the reorder buffer.

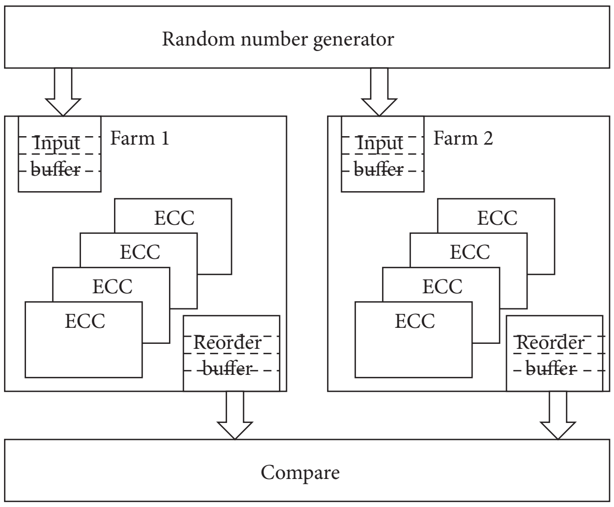

Figure 5: Top level block diagram of the elliptic curve cryptography processor server farm and the test bench.

4.3. Random Number Generator. We reused the public advanced encryption standard (AES) cryptographic algorithm to achieve a NIST-recommended secure random number generator [12] with Verilog HDL. Thanks to Drimer [13], we could utilize the fastest AES Verilog HDL module to make the random number generator. Complex and repetitive permutations facilitate the ciphering algorithm to generate random numbers. According to the cipher theory, encrypted data with enough high security density behave as random numbers because after the repetitive permutation, data retain the characteristics of whitening. We then again created a wrapper of the AES Verilog IP to fit in the BSV test bench program.

The test bench program instantiates the 2 cryptography processor server farms and dispatches the same jobs to each farm. We used fully filled FIFOs to evenly distribute the data elements one at a time as we did in [14]. As expected, the test result showed matched values from two server farms. The elliptic curve cryptography processor server farm accomplished the consecutive jobs in the order the data were received.

5. Conclusions

ESL methodologies have evolved from algorithmic modeling such as architectural explorations and proving concepts as supplementary techniques such as design of embedded systems, system level test bench development, hardware/software cosimulation, and high-level synthesis of ASIC/FPGA designs. To rapidly prototype the server farms, a 193-bit elliptic curve cryptography processor engine with a new suggestion of 3X faster GF multiplier was implemented together with input buffers and reorder buffers each for out-of-order completion. An AES cryptographic function module was reused to produce random number test vectors. Using BSV, we could utilize a higher system level of abstraction than that with traditional HDLs to implement two farms with 4 crypto-processor engines. Since BSV description of the server farms is at very high level, we can modify and explore the architecture with ease. If we parameterize the number of farms and the number of the processor engines, we will get totally different server farms, which will take order of magnitude much time and effort to build the same implementation using Verilog or VHDL.

Due to increasing demand of new technology, the complexity of hardware and software consisting embedded systems is rapidly growing. Now maybe the time that we should shift the hardware/software design paradigm to languages with a higher level of abstraction such as BSV.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] *Bluespec SystemVerilog Reference Guide, Revision 20*, 2011.

[2] S. Moon, J. Park, and Y. Lee, "Fast VLSI arithmetic algorithms for high-security elliptic curve cryptographic applications," *IEEE Transactions on Consumer Electronics*, vol. 47, no. 3, pp. 700–708, 2001.

[3] E. Mastrovito, *VLSI architectures for computation in galois fields [Ph.D. thesis]*, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1991.

[4] R. Afreen and S. C. Mehrotra, "A review on elliptic curve cryptography for embedded systems," *International Journal of Computer Science & Information Technology*, vol. 3, no. 3, p. 84, 2011.

[5] K. Kobayashi, N. Takagi, and K. Takagi, "Fast inversion algorithm in GF(2^m) suitable for implementation with a polynomial multiply instruction on GF(2)," *IET Computers & Digital Techniques*, vol. 6, no. 3, pp. 180–185, 2012.

[6] *SEG 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography Group, September 2000.

[7] I. Koren, "High-speed multiplication," in *Computer Arithmetic Algorithms*, chapter 6, pp. 99–106, Prentice Hall, 1993.

[8] Miran Lipovaca: Learn You a Haskell for Great Good: No Starch Press, 2011.

[9] Guarded Command Language, Wikipedia, http://en.wikipedia.org/wiki/Guarded_Command_Language.

[10] Hoare Logic, Wikipedia, http://en.wikipedia.org/wiki/Hoare_logic.

[11] N. Dave, "Designing a reorder buffer in bluespec," in *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04)*, pp. 93–102, San Diego, Calif, USA, June 2004.

[12] "NIST: NIST-recommended random number generator based on ANSI X9.31 appendix A.2.4 using the 3-key triple DES and AES algorithms," 2005.

[13] S. Drimer, T. Guneysu, and C. Parr, "DSPs, BRAMs, and a pinch of logic: extended recipes for AES on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 1, 2010.

[14] S. Moon, "Design of multiplication server farms using electronic system level with higher level of abstraction," *Advanced Science Letters*, vol. 19, pp. 1441–1444, 2013.