FPGA Offloading for Diffie-Hellman Key Exchange using Elliptic Curves

Dorin-Marian Ionita
Computer Science Department
University Politehnica of Bucharest
Bucharest, Romania
dorin.marian.ionita@gmail.com

Emil Simion
Mathematics Department
University Politehnica of Bucharest
Bucharest, Romania
emil.simion@upb.ro

***Abstract*—Cryptographic offloading to hardware is a hot research topic promising accelerated execution time and improved security compared to the software counterpart. However, hardware design and production is a lengthy process which enquires significant financial resources and technical expertise. Our research paper focuses on elliptic curve cryptography, specifically Diffie-Hellman, and on minimizing these deficiencies by highlighting solutions to map this class of algorithms to hardware description. The insights are not limitative and can be equally applied to other cryptographic primitives.**
The resulting design uses few hardware resources, has low power consumption, is easy to interface with the software and can be implemented on cheap FPGAs.

***Index Terms*—elliptic curves, cryptography, diffie-hellman, FPGA, hardware security, high level synthesis**

I. Introduction

Elliptic curve cryptography has the advantage that it provides a higher level of security than cryptography based directly on finite fields, resulting in keys of smaller sizes for equal protection. As any other algorithm, ECDH (Elliptic Curve Diffie-Hellman) can be implemented either in software or in hardware, each approach having its own benefits and deficits.

When implemented in software, the development time is smaller and therefore the productivity of the programmer is improved. Unfortunately, due to the multi-layered abstraction stack (hardware, programming language, libraries, etc.) and because of the general nature of the underlying hardware (a generic processor) software implementations tend to be slower than the hardware ones [5]. Moreover, algorithms implemented in software have a larger attack surface because the intended interface of the component is not necessarily the only access point to the code and data [6]. An attacker will look for alternative attack paths.

The previous argument holds equally true for hardware, but since it is placed at a lower level of the abstraction stack, the unintended access points are severely limited, especially on dedicated implementations - FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit). While the hardware adds security benefits and improves the execution time, software implementations are often preferred, not only because of the more rapid development process, but also because ASICs and FPGAs demand significant financial investments [7]. Furthermore, it is more difficult to interface hardware with software than software-to-software components, because of the need to match different level of the abstraction stack.

This paper presents the results and insights obtained from our attempts to mitigate the aforementioned downsides of hardware implementation for cryptographic schemes, more specifically applied to ECDH. We start with a brief overview of related topics and existing work, as presented in section II. Next, in section III we explain our development cycle, responsible for reducing the development time. In section IV, we continue by providing details about the high level synthesis implementation of the cryptographic module. In section V and the Appendix, the interface between the PS (Processing System, the CPU) and the PL (Programmable Logic, the FPGA) is explained in more detail. These implementation details, especially those related to HLS (high level synthesis), are directly connected to section VI, which present our practical results: quantity of the hardware resources used from FPGA, execution time, power consumption and timing closure, price of the SoC (System on Chip) we used. Finally, section VII summarizes our results and highlights possible future research direction.

II. Topic Overview and Related Theoretical Work

For our hardware implementation of ECDH we used a number of already well established techniques and theoretical results, which are presented in more detail in this section.

From a theoretical perspective, the Diffie-Hellman Key Exchange [8] scheme is presented. Next, Diffie-Hellman needs a random number generator for the choice of the key and to this aim we used a linear shift register [9]. While the initial description of the algorithm used the multiplicative group of integers modulo n for the underlying mathematical support needed to define the discrete logarithm problem [10], a more secure approach is to use the elliptic curves over a finite field. In fact, using elliptic curves leads to shorter key lengths for the same level of security compared to the classical version [1].

Implementing arithmetic operations on elliptic curves, as ECDH needs, implies a way to compute modulo n inverse, which can be done using the extended version of Euclid's Algorithm, in the iterative version.

From a practical approach, we summarize the HLS technique [11] and its advantages and disadvantages when compared to classical RTL (Register Transfer Level) design.

A. Diffie-Hellman Key Exchange

Public key cryptographic schemes allow 2 parties to share a secret over a public channel. Each party involved in the communication possesses a pair of keys: a public one and a private one. The public key is shared with the other party over the public channel, while the private key is only know to its owner. Combining the public key of the communication partner with one's private key leads to the secret. The obtained secret is the same for both parties.

More to the point, Diffie-Hellman particularizes this scheme in a manner in which obtaining the private key from the public key requires the attacker to solve the discrete logarithm problem [10] (for which no known efficient algorithm is known [10]).

The algorithm requires that the communicating parties previously (perhaps publicly) share a number of parameters: a large primer number p (which together with multiplication defines a finite field $(\mathbb{Z}_p^*, \cdot)$) and an initial integer x such that $2 \leqslant x \leqslant p-1$.

The algorithm is symmetrical for both communication parties, as presented in Algorithm 1, where k_A denotes own's private key, K_A own's public key, K_B the public key of the other party, and $k_{A,B}$ the shared secret.

Algorithm 1 Diffie-Hellman over $(\mathbb{Z}_p^*, \cdot)$

```
INPUT: $p$, $x$
$k_A$ = $random\_choice(\{2, ... p-2\})$
$K_A \equiv x^{k_A} \mod p$
$send(K_A)$
$receive(K_B)$
$k_{A,B} \equiv {K_B}^{k_A} \mod p$
```

B. Pseudo-Random Number Generators

An entropy source or a pseudo-random number generator is needed in order to choose a private key in the Diffie-Hellman scheme. To this aim, we used the theoretical basis of the LFSR (Linear-Feedback Shift Register) [9]. An example scheme for such a register, with degree 3, is presented in Figure 1 [12].

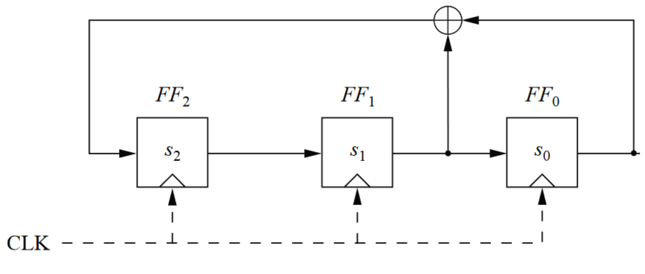


Fig. 1. Linear-Feedback Shift Register, degree 3 [12]

In Figure 1, at each time moment (set by the clock signal), a new value for the s_i bits is computed, as a combination of xor-sum operations on the values of the bits at a previous time moment. The numbers resulting from the concatenation of the bits have good statistical randomness properties over time [9].

As explained in [12], this design of pseudorandom-number generator is straightforward to be extended to arbitrary lengths and we used it to obtain private keys in our implementation of ECDH.

C. Elliptic Curves

As shown in [13], there are algebraic structures which provide better security for Diffie-Hellman Key Exchange than $(\mathbb{Z}_p^*, \cdot)$. More precisely, elliptic curves can be defined over a finite field.

Such an elliptic curve, denoted by $\mathbb{E}(\mathbb{Z}_p)$, is defined as the subset of points from $\mathbb{Z}_p^2$ satisfying equation 1 together with a distinguished ∞ point,

$$y^2 \equiv x^3 + ax + b \tag{1}$$

where $a, b \in \mathbb{Z}_p$ and $0 \not\equiv 4a^3 + 27b^2 \mod p$.

We denote by $\#\mathbb{E}(\mathbb{Z}_p)$ the number of points on $\mathbb{E}(\mathbb{Z}_p)$.

Once the set on which the algorithm operates is defined, we also need to define an equivalent to the multiply operations shown in Algorithm 1.

Let (x_1, y_1), $(x_2, y_2) \in \mathbb{E}(\mathbb{Z}_p)$. We define $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ using formulae 2 and 3, where s is defined by formula 4.

$$x_3 \equiv s^2 - x_1 - x_2 \mod p \tag{2}$$

$$y_3 \equiv s(x_1 - x_3) - y_1 \mod p \tag{3}$$

$$s \equiv \begin{cases} (3x_1^2 + a)(2y_1)^{-1} \mod p & \text{, if } (x_1, y_1) = (x_2, y_2) \\ (y_2 - y_1)(x_2 - x_1)^{-1} \mod p & \text{, otherwise} \end{cases} \tag{4}$$

Finally, let $P \in \mathbb{E}(\mathbb{Z}_p)$ and $n \in \mathbb{N}$, then we define addition of a point with itself n times as in formula 5.

$$nP = \begin{cases} P + (n-1)P & \text{, if } n \geq 2 \\ P & \text{, if } n = 1 \end{cases} \tag{5}$$

One last theoretical result, related to elliptic curves, which we used in our hardware implementation is Hasse's Theorem (formula 6) [14]. It provides a boundary for $\#\mathbb{E}(\mathbb{Z}_p)$.

$$p + 1 - 2\sqrt{p} \leqslant \#\mathbb{E}(\mathbb{Z}_p) \leqslant p + 1 + 2\sqrt{p} \tag{6}$$

D. Algorithm for Computing Inverse modulo n

As show in formula 4, an algorithm for computing inverses modulo n is needed in order to implement arithmetic operations on elliptic curves. One example of such an algorithm is presented in [15], based on extended Euclid's Algorithm.

E. Diffie-Hellman Key Exchange using Elliptic Curves

Using the definitions in section II-C, algorithm 1 can be adapted to have $(\mathbb{E}(\mathbb{Z}_p), \cdot)$ as underlying support. This adaption, known as ECDH is presented in Algorithm 2.

This time, the hyperparameters of the algorithm are the curve parameters a and b, the primitive element $G \in \mathbb{E}(\mathbb{Z}_p)$ and the prime p.

Algorithm 2 Diffie-Hellman over $(\mathbb{E}(\mathbb{Z}_p), \cdot)$

```
INPUT: $a$, $b$, $p$, $G$
$k_A = random\_choice(\{2, ...\#\mathbb{E}(\mathbb{Z}_p) - 1\})$
$K_A = k_A G$
$send(K_A)$
$receive(K_B)$
$k_{A,B} = k_A \mathrm{K}_B$
```

F. High Level Synthesis

For our hardware design we used a mixed approach of RTL and HLS design. This is because we aimed at accessing the advantages of both approaches, while eliminating the deficiencies.

The usual approach to hardware design for FPGA is to use low level hardware description languages such as VHDL and Verilog. While these languages provide good control of the synthesized design, the drawback is that development at this abstraction level, called RTL, is a slow process.

To mitigate this, HLS allows the hardware designer to use a high level hardware description language, usually based on a programming language such as C, C++, Scala or Haskell [16]. The hardware described using such a language is then compiled to a low level hardware description language which, in turn, can be synthesized to program a FPGA.

III. Hardware-Software Codesign Cycle

In our implementation we took a multi-step approach to our development, as presented in Figure 2.

First, ECDH is implemented in Vivado HLS based on C. Once the implementation is complete, it is considered software and compiled to binary, followed by testing against a software test bench. Once the functional tests are passed, the HLS description is compiled to VHDL. Multiple target frequencies are tried, in descending order, and the HLS description is optimized until time closure is achieved. The final HLS version is compiled to an RTL description and is validated against the same test bench.

The resulting module is then integrated with the rest of the SoC at RTL level. This is where the interconnecting between the PS and the PL is described, as well as where the maximum possible frequencies are set. After synthesis and after the FPGA is programmed, the drivers, which run on the PS, are written. They are needed to send inputs to the FPGA and to receive outputs from it, and for performance measurements.

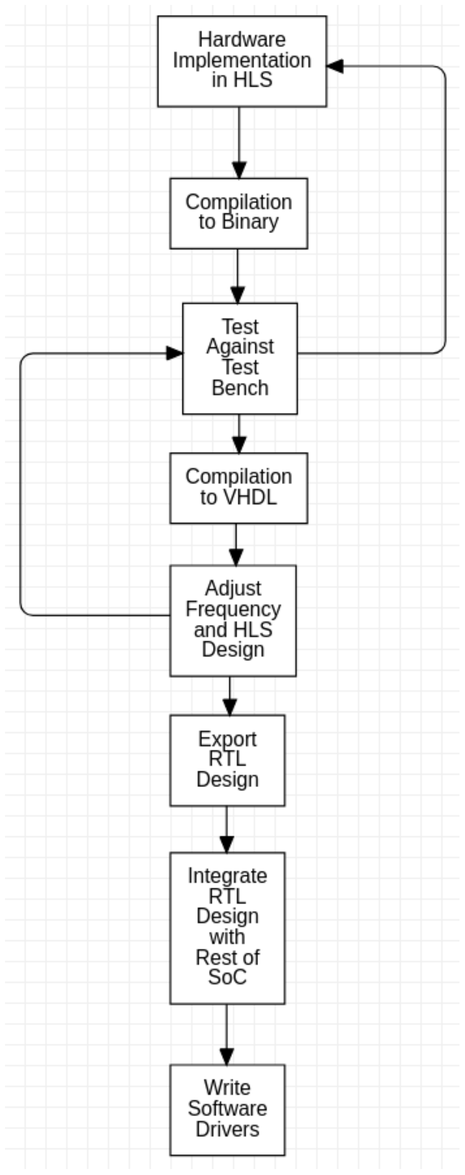


Fig. 2. Hardware-Software Codesign Cycle

IV. High Level Synthesis Architecture Details

In the first part of this section, we describe our architecture as a black-box, explaining the desired behaviour and the interface to the processing system.

In the following parts of this section, we follow the white-box approach to explain the actual details of the HLS implementation. We show how we considered the limitations and enhancement of the HLS language compared to the C programming language, when applied to ECDH: arbitrary size data types and associated arithmetic operations, lack of recursion or arbitrary length loops, lack of complex mathematical operations, persistent and secure data storage, external interfaces.

A. The Accelerator as Black-Box

For the High Level Synthesis, we developed the IP (Intellectual Property) in Figure 3. Ports $s\_axi\_CONTROL\_BUS$, ap_clk, ap_rst_n and $interrupt$ have meaning only in the context of interconnection with the PS and are explained in section V, while the rest of them come from our implementation of ECDH, and are detailed in the next 2 paragraphs.

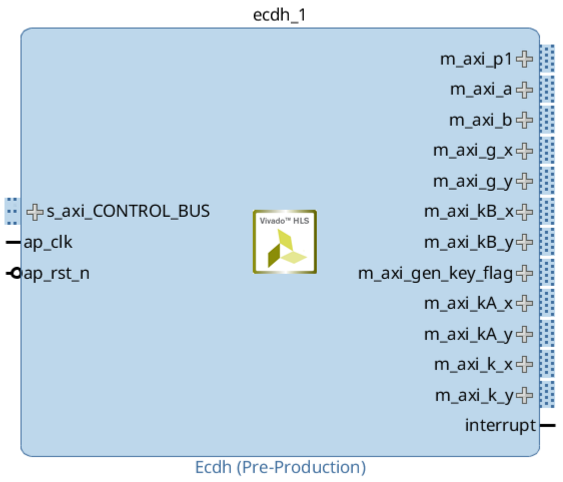


Fig. 3. ECDH IP Generated Using HLS

The IP works in 2 stages. In the first one, the private key is generated (and kept inside the FPGA). The output of this stage is the public key. The PS sets the curve parameters: $p1$ (the prime number), a and b (the curve coefficients) and g_x and g_y (the primitive element). Signaling that we wish to generate the key (as opposed to the shared secret) is done by setting the gen_key_flag. After the execution of this step, the public key is available on the kA_y and kA_x ports.

In the second step, the public key of the communication partner is sent to the FPGA using the ports kB_x and kB_y, and the gen_key_flag is not set. The resulting shared key will be available on ports k_x and k_y.

In the HLS language, the inputs and outputs of the black-box are described using specific pragma directives, as shown in Listing 1.

```
#pragma HLS INTERFACE s_axilite port=return
    bundle=CONTROL_BUS
#pragma HLS INTERFACE m_axi port=p_in
    offset=slave bundle=p
```

Listing 1. Pragma Directives Describing the Interface

B. Arbitrary Data Types and Associated Operations

The most obvious difficulty in implementing the algorithm in hardware is that we need to use large numbers and perform basic arithmetic operations on them. In a programming language one could use a large numbers library and in Verilog/VHDL vectors with arbitrary number of elements, each with 1 bit length can be employed. These approaches are not available in HLS. Fortunately, the custom hardware description language defines arbitrary size data types. One such case is our definition of the curve parameters, as shown in Listing 2. For example $uint224$ defines a 224 bits long unsigned integer. The unsigned integer type specification is important because there are a number of operators defined on it: addition, multiplication, subtraction, division and modulo. Pointer referencing and dereferencing for this type is also possible.

```
uint224 a, b;
```

Listing 2. Arbitrary Size Curve Parameters in HLS

Furthermore, using this abstraction from HLS, operations on smaller data type are implicitly implemented (provided the user fills the remaining bits with 0). This allows a hardware design with a higher level of generality, for example a 256 key-length ECDH implementation will also support a 224 bits one, ultimately resulting in shorter development time for the hardware.

C. Mitigating Lack of Recursion and Arbitrary Length Loops

Notably, an operation which is not defined for arbitrary length data types is the modulo inverse. In order to implement this operation, we used the extended version of Euclid's Algorithm. Since recursion can not be used in HLS, we had to use the algorithm in iterative form, as explained in [15]. Moreover, loops of length that can not be known at compile time, such as $while$ instructions, can not be used in HLS. To mitigate this, we simply used a for instruction of maximum possible length for the data type and a $break$ instruction for stopping the loop earlier. The precise code is shown in Listing 3.

```
  for (i = 0; i <= max_uint224; i++) {
    if (b1 == 0)
      break;
    ...
  }
```

Listing 3. Persistent and Secure Data Storage

D. Mitigating Lack of Complex Mathematical Operations

With respect to generation of the private key, Algorithm 2 requires a random choice of a number based on the number of the points on the elliptic curve. In order to gain the number of points we started from the prime parameter of the curve and Hasse's Theorem (briefly explained in section II-C). The disadvantage of this theorem is that it requires a square root computation, which can be costly to do in FPGA. A simpler approach is to find a lower bound of the number of points on the elliptic curve and use it instead of the square root.

Starting from expression 6, and considering (from Algorithm 2) $k_A \leqslant \#\mathbb{E}(\mathbb{Z}_p) - 1$, equation 7 follows.

$$k_A \leqslant p + 1 - 2\sqrt{p} - 1 \tag{7}$$

Equation 7 can be simplified to equation 8.

$$k_A \leqslant p - 2\sqrt{p} \quad (8)$$

For the next step of our boundary proof, we notice that for large enough p expression 9 holds true.

$$\sqrt{p} \leqslant \frac{p}{4} \quad (9)$$

From 8 and 9 the boundary in formula 10 follows, which allows us to choose the private key of ECDH randomly from the set $\{0..p/2\}$.

$$k_A \leqslant \frac{p}{2} \quad (10)$$

E. Persistent and Secure Data Storage

In order to further improve the security of our design, the private key never leaves the FPGA. To achieve this in HLS, it suffices to define the variable representing the key as global and static - the static keyword in this case actually signifies that the value is kept between multiple calls to the accelerator. This exact technique was also used to avoid resetting the LSFR at private key generation. A code snippet for this is shown in Listing 4.

```
static uint224 lfsr;
static uint224 private_key;
```

Listing 4. Persistent and Secure Data Storage

V. Register Transfer Level Design

The interconnection between the PS and the PL was implemented at RTL level, being based on the AMBA [17] bus protocol and a double master-slave approach, as shown in the Appendix on the last page of this paper.

We used 2 AXI [18] Interconnect modules from Xilinx to buffer the data between our accelerator and the PS. The whole system lays in a single clock domain controlled by the PS. The Processor System Reset modules offers the PS the ability to control the PL.

The data and control ports of the accelerator are memory-mapped and can be accessed from the accelerator just as any other memory region.

VI. Experimental Setup and Results

A. Execution Environment

The hardware on which we ran our experiments is a Zynq7000-based board [19], which as of 2021 costs less than 500$ [20]. For testing purposes, we implemented the drivers for our accelerator on top of FreeRTOS [23], mainly by reusing the code from the test benches in the codesign phase. We used software timers to measure the execution time. The precise elliptic curve we used for our experiments is Weierstraß P-224, as standardized by NIST [21].

Resource	Utilization	Available	Utilization %
LUT	17940	53200	33.72
LUTRAM	391	17400	2.5
FF	27697	106400	26.03
BRAM	36.50	140	26.07
DSP	0	220	0
BUFG	1	32	3.13

TABLE I
FPGA Consumed Resources

B. Timing, Execution Speed, Hardware Resources and Power Consumption

The hardware resources utilized from the FPGA are presented in table I.

With respect to timing closure, we managed to obtain stability for our accelerator at a frequency of 92 MHz, resulting in similar execution speed for both steps of our accelerator (as presented in section IV-A).

The execution time is heavily dependent on how large the randomly chose private key is. In our experiments we noticed that it varies from 431 ms to 1.309 s.

The power consumption of our accelerator is estimated to 1.832W by Vivado [22].

VII. Conclusions and Further Work

For the elliptic curve cryptography class of algorithms, we have shown how using HLS in conjunction with RTL level design, for FPGA, can lead to few hardware resources consumption, energy efficiency and quick development time.

We have highlighted specific approaches to mapping ECDH to hardware and these approaches can be used for other cryptographic algorithms as well, even on cheap FPGAs.

Furthermore, we have shown that the codesign cycle improves the interfacing between the software driver and the hardware accelerator.

The execution speed can be improved and more research is needed in this direction. We acknowledge that limiting the possible choices for the private key, while required because of the HLS limitations, is not desirable from a security perspective. More theoretical research is needed in order to obtain a larger bound for the maximum value of the private key, thus improving security.

References

[1] Senekane, Makhamisa Qhobosheane, Sehlabaka & Taele, Benedict - Elliptic Curve Diffie-Hellman Protocol Implementation Using Picoblaze, 2011

[2] Zerene Sangma, Hardware implementation of elliptic curve Diffie-Hellman key agreement scheme in GF(p), 2018

[3] Muhammad Malik, Efficient Implementation of Elliptic Curve Cryptography Using Low-power Digital Signal Processor, 2010

[4] Ionita, Dorin & Manole, Filip & Slusanschi, Emil - Efficient Parallel Simulatin of Wireless Signal Propagation, 2020

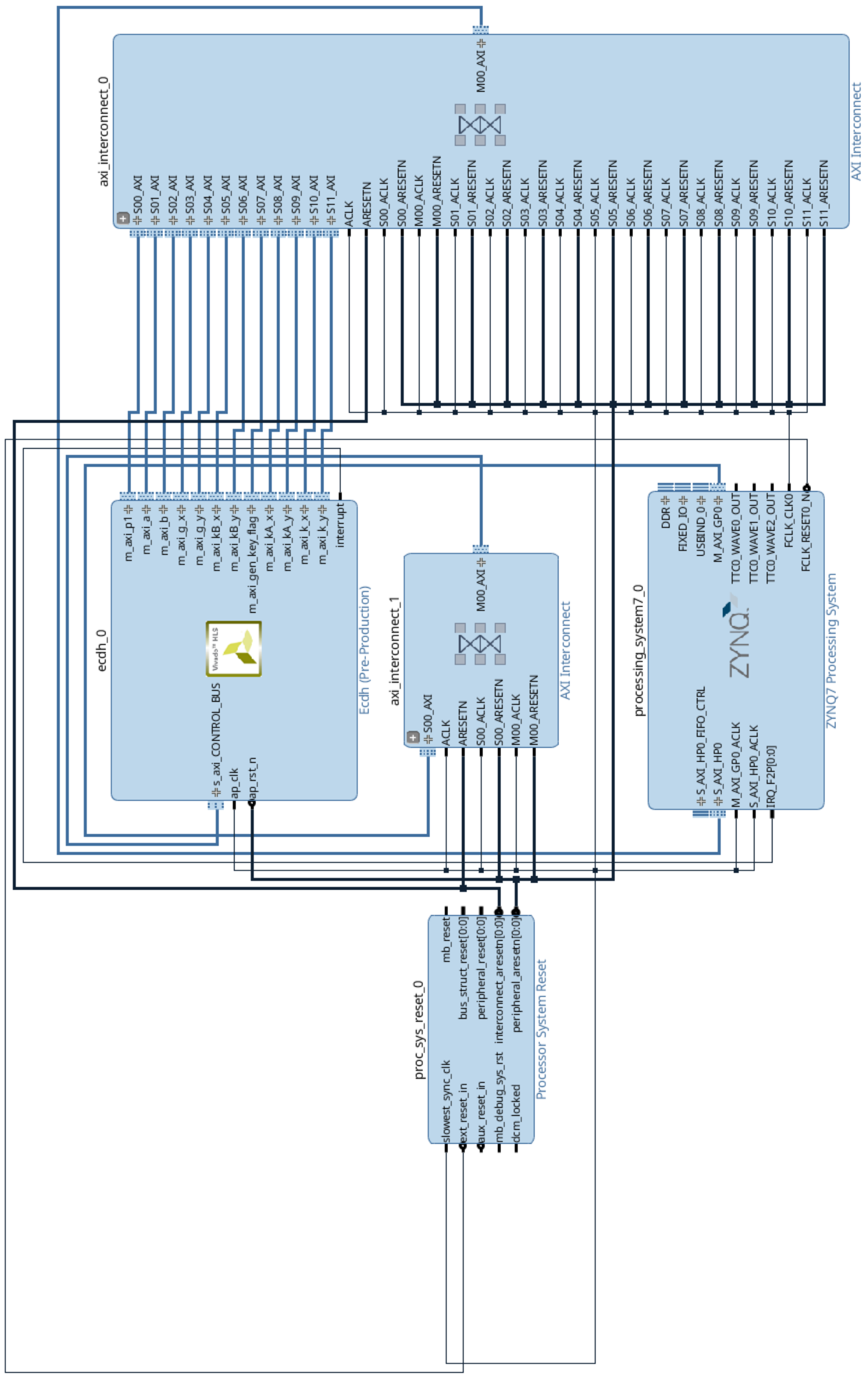
[5] Ionita, Dorin & Deaconescu, Razvan & Chiroiu, Mihai - Hardware Implementation of the Python Virtual Machine, 2019

[6] Aleph, One - Smashing The Stack For Fun And Profit, Phrack, 1996

[7] Intel - Decrease Total System Costs with Industry's Lowest Cost, Lowest Power FPGAs, 2019

[8] Kar, Jayaprakash & Mishra, Manoj Ranjan - A study on diffie-hellman key exchange protocols, 2017
[9] Shabaz, Mohammad & Patel, Anand, et. all - Design of Reconfigurable 2-D Linear Feedback Shift Register for Built-In-Self-Testing of Multiple System-on-Chip Cores, 2015
[10] Corrigan-Gibbs, Henry & KoganThe, Dmitry - Discrete-Logarithm Problemwith Preprocessing, 2019
[11] Meeus, Wim & Van Beeck, Kristof et. all - An overview of today's high-level synthesis tools, 2012
[12] Paar, Cristoph & Pelzl, Jan - Understanding Cryptography, pp. 40-44, 2010
[13] Amara, Moncef & Siad, Amar - Elliptic Curve Cryptography and its applications, 2011
[14] Soeten, Mirjam - Hasse's Theorem on Elliptic Curves, 2018
[15] Liu, Chao-Liang & Horng, Gwoboa & Liu, Hsin-Yu - Computing the modular inverses is as simple ascomputing the GCDs, 2007
[16] Selvaraj, Henry & Daoud, Luka & Zydek, Dawid - A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing, 2013
[17] Suan, Wang Huan & Jambek, Asral Bahari - Design and analysis of microcontroller system using AMBA-Lite bus, 2017
[18] Neelavathi, D. - Design of AMBA 3.0 (AXI) Bus based SoC, 2012
[19] Tang, Zhiwei - The Zynq-7000 SoC on UltraScale Architecture, 2021
[20] Kumar, Santosh & Shah, Madhu & Singh, Arjun - FPGA – Raspberry pi Interface for low cost IoT based image processing, 2017
[21] Chen, Lily & Moody, Dustin et. all. NIST: Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters, 2019
[22] Feist, Tom - Vivado Design Suite, 2012
[23] Zhu, Ming-Yuan - Understanding FreeRTOS: A Requirement Analysis, 2011

Appendix
RTL Interconnect Between the Processing System and the Programmable Logic


axi_interconnect_0
S00_AXI
S01_AXI
S02_AXI
S03_AXI
S04_AXI
S05_AXI
S06_AXI
S07_AXI
S08_AXI
S09_AXI
S10_AXI
S11_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
S01_ACLK
S01_ARESETN
S02_ACLK
S02_ARESETN
S03_ACLK
S03_ARESETN
S04_ACLK
S04_ARESETN
S05_ACLK
S05_ARESETN
S06_ACLK
S06_ARESETN
S07_ACLK
S07_ARESETN
S08_ACLK
S08_ARESETN
S09_ACLK
S09_ARESETN
S10_ACLK
S10_ARESETN
S11_ACLK
S11_ARESETN
M00_AXI
AXI Interconnect
ecdh_0
m_axi_p1
m_axi_a
m_axi_b
m_axi_g_x
m_axi_g_y
m_axi_kB_x
m_axi_kB_y
m_axi_gen_key_flag
m_axi_kA_x
m_axi_kA_y
m_axi_k_x
m_axi_k_y
interrupt
s_axi_CONTROL_BUS
ap_clk
ap_rst_n
Vivado HLS
Ecdh (Pre-Production)
axi_interconnect_1
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M00_AXI
AXI Interconnect
processing_system7_0
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
TTC0_WAVE0_OUT
TTC0_WAVE1_OUT
TTC0_WAVE2_OUT
FCLK_CLK0
FCLK_RESET0_N
S_AXI_HP0_FIFO_CTRL
S_AXI_HP0
M_AXI_GP0_ACLK
S_AXI_HP0_ACLK
IRQ_F2P[0:0]
ZYNQ
ZYNQ7 Processing System
proc_sys_reset_0
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset