BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ASYMMETRIC-KEY CRYPTOGRAPHY IN EMBEDDED SYSTEMS

ASYMETRICKÉ KRYPTOGRAFICKÉ ALGORITMY VE VESTAVĚNÝCH SYSTÉMECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATEJ ZÁHORSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. SVETOZÁR NOSKO**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

148408

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Záhorský Matej, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Cybersecurity
Title:	**Asymmetric-Key Cryptography in Embedded Systems**
Category:	Security
Academic year:	2022/23

Assignment:

1. Study the literature on asymmetric cryptography. Focus on asymmetric cryptography suitable for implementation in embedded systems.
2. Select a suitable platform and algorithms suitable for implementation. Consider computational complexity and memory requirements of the algorithms.
3. Implement selected algorithms and discuss the achieved results.
4. Implement and demonstrate data protection (for example, image or sensor data).
5. Summarize the results achieved and discuss possible improvements.

Literature:

- According to the supervisor's instructions.

Requirements for the semestral defence:
Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Nosko Svetozár, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	31.10.2022

Abstract

The primary goal of this thesis is to find and implement an existing asymmetric cryptographic algorithm in a FPGA and evaluate its performance. The first chapter of this thesis focuses on embedded systems and FPGAs, while describing their structure and purpose. The second chapter compares different cryptographic algorithms and their properties, which would make them usable in embedded systems. The design and implementation phases of this project describe and implement a solution, which includes the selection and integration of a signature algorithm in a FPGA. Additionally, further optimizations to increase the computing performance are also implemented in a form of hardware acceleration, which are then compared to the original algorithm in the evaluation chapter.

Abstrakt

Účelom tejto práce je prieskum a implementácia existujúceho asymetrického kryptografického algoritmu v FPGA a vyhodnotenie jeho výkonu. Prvá kapitola sa zameriava na vstavané systémy a FPGA, pričom popisuje ich štruktúru a použitie. V druhej kapitole je porovnanie kryptografických algoritmov a ich vlastností, ktoré umožňujú ich použitie vo vstavaných systémoch. Fázy návrhu a implementácie v tomto projekte popisujú a implementujú riešenie, ktoré zahŕňa výber a integráciu podpisovacieho algorithmu v FPGA. Dodatočné optimalizácie na zvýšenie výkonu sú taktiež naimplementované vo forme hardvérovej akcelerácie, ktoré sú zároveň porovnané s pôvodným algoritmom v kapitole vyhodnotenia.

Keywords

FPGA, cryptography, data signing, hardware acceleration, ECDSA, Microblaze, MbedTLS, pseudorandom number generator, entropy, HLS, Karatsuba

Kľúčové slová

FPGA, kryptografia, podpisovanie dát, hardvérová akcelerácia, ECDSA, Microblaze, MbedTLS, generátor pseudonáhodných čísel, entropia, HLS, Karatsuba

Reference

ZÁHORSKÝ, Matej. *Asymmetric-Key Cryptography in Embedded Systems.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Svetozár Nosko

Rozšírený abstrakt

Úvod

V dnešnej dobe je veľkým trendom Internet of Things (IoT), v ktorom hrá zabezpečenie zariadení veľkú rolu. Tieto zariadenia majú väčšinou obmedzené výpočetné prostriedky, čo im znemožňuje vykonávať zložité kryptografické algoritmy dostatočne efektívne. Cieľom tejto práce je nájsť a naimplentovať podpisovací algoritmus na týchto zariadeniach tak, aby mohol byť vykonávaný v reálnom čase pre relatívne veľké množstvo dát.

Táto práca je zároveň súčasťou rozsiahlejšieho projektu `SECUSEN`, ktorého účelom je zabezpečiť dáta z kamerového senzoru čo najbližšie k ich zdroju, ktorým je FPGA. Z tohto dôvodu je súčasťou riešenia prieskum možností implementácie podpisovania v prostredí FPGA a jeho vyhodnotenie. Na základe výsledkov sa dodatočne vykoná optimalizácia vo forme hardvérovej akcelerácie, ktorá umožní efektívnejšie generovanie podpisov.

Návrh

Prvým dôležitým krokom je vyhľadanie vhodného podpisovacieho algoritmu, ktorý by bol použiteľný na systéme s obmedzeným výkonom a pamäťovými prostriedkami. Vhodným kandidátom je algoritmus používajúci eliptické krivky ECDSA, ktorý disponuje menšími veľkosťami použitých kľúčov, čím znižuje požiadavky na pamäť. Implementácia tohto algoritmu je dostupná v softvérovej knižnici MbedTLS, ktorá sa zameriava na kryptografické algoritmy pre vstavané systémy s architektúrou ARM. Konfigurácia použitých modulov v tejto knižnici umožňuje markantné zredukovanie veľkosti kódu na menej ako 100 KB (viď. 6.6). Dodatočne je možné vybrať z niekoľkých dostupných typov eliptických kriviek, ktoré sa vyhodnotia a porovnajú pred ich použitím vo finálnom riešení.

Dôležitou súčasťou knižnice MbedTLS je však podpora výpočetnej jednotky Microblaze, ktorú možno syntetizovať a spustiť na FPGA. Tento procesor je konfigurovateľný pre špecifické požiadavky, ktoré sú dôležité pre efektívne vykonávanie algoritmu ECDSA. Dostupnosť vlastností, ako je násobička alebo delička celých čísel môžu mať veľký vplyv na rýchlosť výpočtu modulárneho umocňovania, na ktorom je založený tento podpisovací algoritmus.

Bezpečnosť algoritmu ECDSA však do veľkej miery závisí na náhodnosti, ktorá však nie je priamo dostupná a musí byť naimplementovaná z externého zdroja. Algoritmus PCG je vhodný na generovanie pseudonáhodných čísel v algoritme ECDSA pre jeho dobré štatistické vlastnosti a nízke nároky [19]. Jeho počiatočný stav však musí byť správne inicializovaný náhodnou entropiou, ktorú možno zozbierať pomocou analógovo-digitálneho prevodníka (ADC). Navrhovaným prístupom je zber niekoľkých záznamov z teplotného senzoru, ktoré spoločne vygenerujú výslednú entropiu.

Jednou z možností pre dosiahnutie vyššieho počtu vygenerovaných podpisov za sekundu je optimalizácia pomocou Merklovho stromu. Táto štruktúra umožnuje ukladanie vygenerovaných hashov do binárneho stromu, z ktorých sa vytvorí jeden koreňový hash určený na podpísanie. Na základe množstva uložených hashov sa tak zredukujú požiadavky na rýchlosť algoritmu ECDSA, čím sa zvýši jeho použiteľnosť v reálnom čase.

Tento prístup však nie je jedinou možnosťou v FPGA. Syntéza vyššieho levelu (HLS) umožňuje implementáciu algoritmu v jazyku C, z ktorého sa vygeneruje popis v jazyku HDL a následne sa vysyntetizuje hardvérový blok vykonávajúci túto funkciu. Tento blok

sa nakoniec zaintegruje do softvéru MbedTLS ako hardvérový akcelerátor, ktorý má potenciál zvýšiť efektivitu podpisovania. Na nájdenie vhodného kandidáta pre akceleráciu je možné využiť profilovacie nástroje, ktoré zobrazujú čas strávený v jednotlivých funkciách algoritmu.

Implementácia

Navrhované riešenie bolo naimplementované na FPGA prítomnom na zariadení Zynq ZC702, ktoré je možné naprogramovať a profilovať z vývojového prostredia Vivado Design Studio.

Z implementačného hľadiska došlo k zmenám oproti návrhu len v použitom generátore pseudonáhodných čísel. Generátor PCG podľa štúdie [20] nie je kryptograficky bezpečný a nemal by byť použitý v bezpečnostných aplikáciach. Z tohoto dôvodu bol ako náhrada vybraný generátor ChaCha20, ktorý sa automaticky aktualizuje po uplynutí niekoľkých minút alebo po vygenerovaní určitého počtu bytov.

Na vytvorenie akcelerátoru bol vybraný algoritmus pre modulárne umocňovanie, ktorý podľa profilovacích nastrojov využíval najviac procesorových cyklov. Násobenie bolo implementované pomocou Karatsubovho algoritmu, ktorý vstupné 256-bitové hodnoty dekomponuje na 128-bitové časti. Tieto 4 hodnoty sú navzájom vynásobené pre získanie 512-bitového výsledku, avšak použitím optimalizovanej funkcie bol počet násobení zredukovaný zo 4 na 3, čo znížilo celkovú latenciu obvodu. Niektoré operácie naopak vyžadovali úpravu algoritmu zvýšením latencie, aby došlo k zníženiu množstva použitých DSP48 blokov, inak by nebolo možné tento akcelerátor umiestniť na FPGA.

Vyhodnotenie

Na začiatku implementačnej fázy bolo nutné vyhodnotiť výkon podpisovania ECDSA na Microblaze. Z mnohých dostupných rozšírení boli vybrané hardvérové bloky pre násobenie, bitové posuvy a predikciu skokov, ktoré dohromady priniesli niekoľkonásobný nárast výkonu (viď. 6.2).

Z hľadiska podpisovacieho algoritmu ECDSA bolo nutné vybrať vhodnú eliptickú krivku na základe jej výkonu a pamäťových požiadavkov. Z troch vhodných kandidátov mala eliptická krivka `SECP256K1` najlepšie výsledky (viď. 6.3).

Po implementácii hardvérového akcelerátoru na modulárne umocňovanie bolo potrebné vyhodnotiť jeho výhody a nevýhody. Čas podpisovania sa z pôvodných 88 ms znížil na 33 ms (podľa použitej optimalizácie kompilátoru), čo vo výsledku dáva 2.5x zrýchlenie. Na druhú stranu došlo k zvýšeniu pamäťových požiadavkov, ktoré znemožnili použitie 64 KB bloku pre všetky levely optimalizácie kompilátoru okrem `-Os` (viď. 6.8).

Na koniec sa vykonalo vyhodnotenie overovania podpisov ECDSA, ktoré po hardvérovej akcelerácii prinieslo podobné zrýchlenie ako algoritmus podpisovania (viď. 6.10 a 6.11).

Záver

Na základe výsledkov vyhodnotenia možno usúdiť, že výsledná implementácia je použiteľná v reálnom čase pre podpisovanie dát z kamery, ktorá produkuje 20 až 30 snímkov za sekundu. Ďalšie zlepšenia a optimalizáce by boli možné použitím Edwards kriviek a algoritmu EdDSA, ktoré majú potenciál znížiť celkový čas podpisovania [8].

Asymmetric-Key Cryptography in Embedded Systems

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Svetozár Nosko. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matej Záhorský
May 15, 2023

Acknowledgements

I would like to express my sincere gratitude to Ing. Svetozár Nosko for his time and will to give me advice and feedback that have led this thesis to a successful conclusion. Additionally, I would like to thank Ing. Petr Musil and Ing. Martin Musil, PhD. for their useful insights into the world of FPGAs, which were essential during the implementation stage of this thesis.

Contents

Chapter 1

Introduction

One of the most recurring problems of modern technology has always been data security and authenticity, whether its our personal data sent to a remote server (eg. instant messaging) or more sensitive (secret) data within a corporation. Nowadays, more and more attention is being given to Internet of Things (IoT), where a series of microcontrollers communicate within a local network to create an ecosystem. One of the most common examples is smart home, where each device provides some sort of functionality to the user (eg. lighting control, heating, ventilation). Information from this system is mostly in the local network, but it can also be accessed remotely, which poses a significant vulnerability to the user. For this reason, the data itself and access to this data has to be sufficiently protected, otherwise we would not be able to trust the information we receive or even worse, we would lose the control of our own system.

This project focuses on the former of the two and to achieve this, data created on a microcontroller (gathered from a sensor) must be signed before they leave the device. Generally, this task is often done on a master device, where all of its slave devices are interconnected and sending their raw data, which poses a risk of forgery. This has been the case mostly due to insufficient resources, so cryptographic algorithms could not simply be processed efficiently and quickly enough. Nowadays, many new microcontrollers provide hardware acceleration for such algorithms, which greatly improves the devices performance by offloading these computation-heavy tasks.

Many modern sensors that provide some kind of data source are designed in Field Programmable Gated Arrays (FPGAs). This fact presented an opportunity to research and design a system, which could also properly secure the data as soon as they are produced to ensure their authenticity and integrity. Additionally, this project is a part of a larger project `SECUSEN`, which focuses on utilizing both FPGA and Linux-based systems to create a fully secure ecosystem that can only produce authentic data.

One of the key parts of this project is to research existing cryptographic algorithms for data signing, finding possible ways of implementing them in a FPGA and evaluate the solution to determine its usability, resource usage and performance. Additional optimizations in form of specialized algorithms to further improve computing capabilities of a FPGA are also an important part of this thesis.

Chapter 2

FPGA and embedded systems

The first chapter explains the meaning of a FPGA and describes its usage and structure to better understand its benefits and limitations. The process of hardware design, testing and implementation is also described here in order to see the different challenges in comparison with classic software development. Some specific intellectual properties (IPs), that will be used in the design and implementation phases are mentioned in the last sections of this chapter.

2.1 Embedded system

An embedded system is a system designed for a specific, simple task to keep its price and size as low as possible. Its usually made of an embedded processor and external logical or mechanical components, which the processor reacts to and produces a response [10]. Additional resources such as RAM, storage or input and output interfaces can also be integrated on the chip to create a microcontroller with more advanced computational capabilities. Such additions make the device more usable as it is now capable of communication by using standardized interfaces such as USB or Ethernet. A system on a chip (SoC) has more advanced hardware blocks integrated within its system, such as hardware accelerators[1] made for increasing the computing performance or even more complex circuits like FPGAs[2].

2.2 FPGA

The Field Programmable Gate Array (FPGA) is an integrated circuit often used for implementation, validation and testing of designed hardware blocks before they are physically mass produced. FPGAs are a good alternative to Application Specific Integrated Circuits (ASICs) as the final solution can be incrementally upgraded and tested before production reducing development time and financial costs. In contrast, the performance of an FPGA is generally lower and the printed circuit board (PCB) area requirements along with power consumption are higher than those of an ASIC [2], so this compromise must be taken into account before making a decision between the two technologies.

Generally, FPGAs contain an array of configurable logic blocks (CLBs), interconnects that can be used to wire them together and I/O pads for data transfers to and from the

[1] https://www.st.com/en/microcontrollers-microprocessors/stm32mp1-series.html
[2] https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

FPGA as a means of interaction [11]. CLBs can be used to combine simpler logic gates and memory blocks to create more complex functions, which allow for easier design changes and testing similar to the incremental programming approach in software development. A basic FPGA architecture consisting of these elements forms a matrix, which can be seen in the next figure:

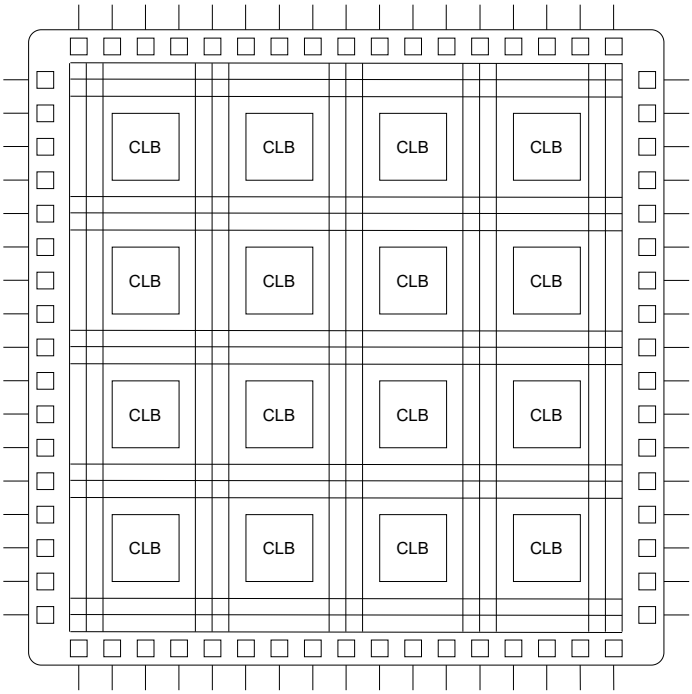

Figure 2.1: Basic FPGA architecture[3].

Based on the FPGA model and manufacturer, a CLB consists of several memory based logic cells known as a cluster [11] as shown in the next figure:

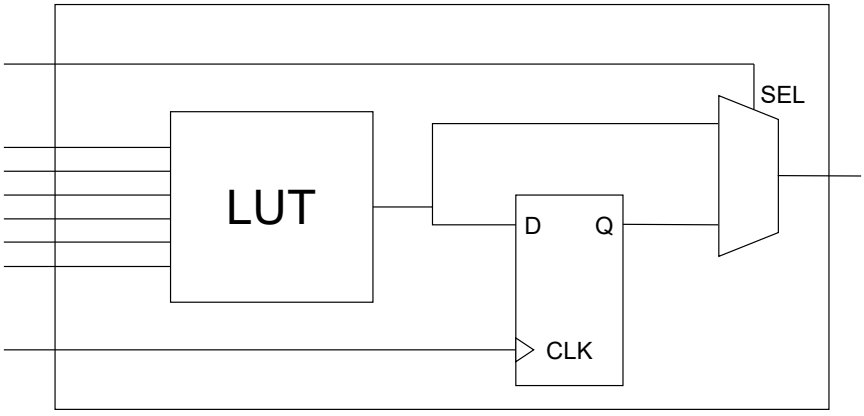

Figure 2.2: Memory based logic cell[4].

[3]Understanding FPGA Architecture. Xilinx Inc. [online]. [cit. 2023-05-10]. Based on: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/odz1504034293215.html

[4]FPGA Logic Cells and Architecture. Haibo Wang. [online]. [cit. 2023-05-10]. Based on: https://www.engr.siu.edu/haibo/ece428/notes/ece428_logcell.pdf

A logic cell usually consists of two types of basic elements:

- Lookup tables (LUTs) – Basic building blocks for implementing logic operations that consist of several levels of interconnected multiplexers and attached memory cells. A generic N-way LUT has N inputs that can yield a desired output from 2^N connected memory cells. A simple 2-way LUT can be seen on the following figure.

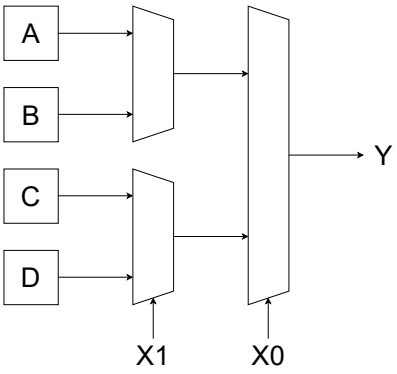

Figure 2.3: 2-way Lookup table with 4 connected memory cells[5].

- D-type flip-flop (DFF) – A register that contains a clock, reset, clock enable, input and output data pins that are used to transfer the value once both the clock enable and clock signals are set to high. This behavior is often useful for memory storage and the outputs of LUTs are often stored in these flip-flop registers.

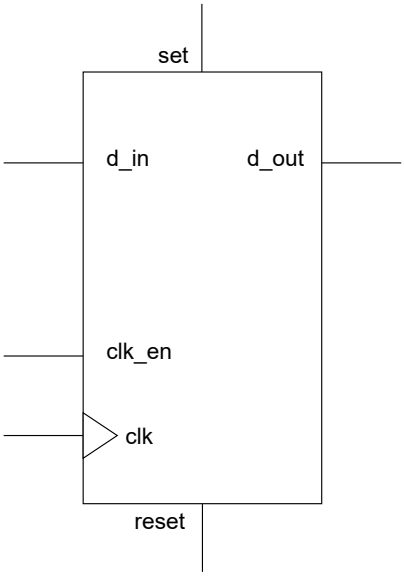

Figure 2.4: D-type flip-flop[6].

[5]LUT. Xilinx Inc. [online]. [cit. 2023-05-08]. Based on:
https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html

[6]Flip Flop. Xilinx Inc. [online]. [cit. 2023-05-08]. Based on:
https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/ksg1504034293914.html

One of the most complex computational blocks that are available on FPGAs is called DSP48. It is a form of an arithmetic logic unit (ALU) composed of a chain of three different blocks. It contains an addition/subtraction unit called pre-adder connected to a 25x18 multiplier, which is in turn connected to another addition/subtraction/accumulation engine. The structure of a DSP48 unit is shown on the following figure:

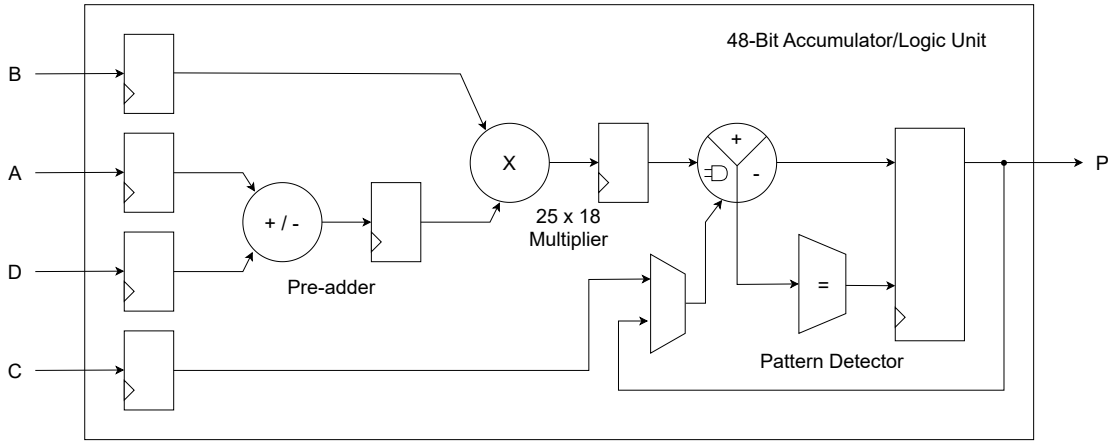

Figure 2.5: DSP48 block[7].

2.3 Hardware design

The hardware block design process is usually executed in following steps:

- Hardware description – Based on the purpose of the block, its inputs, outputs and operations with data are defined by the use of a dedicated programming language, such as VHDL or Verilog.
- Synthesis – The created description is then verified and synthesized for the selected FPGA. The resulting package can be loaded onto the board and executed.
- Debugging and profiling – If the execution of the synthesized design is not performing as expected, the debugging tools available on most platforms allow for easier detection of the issue and can lead to better optimization.

2.4 VHDL and Verilog

VHDL and Verilog are some of the best known hardware description languages (HDL) used for designing integrated circuits including logic gates, multiplexers, latches or memory blocks. Every hardware block is defined as an entity with several input and output signals. The most important signals for ensuring correct operation of each designed block are clock and reset. The reset signal is used to restore the block to its default state, usually before a new task is going to be processed. The clock signal is used for synchronization and it determines when input signals should be read, processed and stored onto a specified output.

[7]DSP48 Block. Xilinx Inc. [online]. [cit. 2023-05-08]. Based on:
https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html

All blocks present in the final design are executed in parallel and they must either have a matching input clock frequency or a specialized circuit for clock domain crossing (as described in [22]) to avoid producing incorrect results. The difference in frequency would cause the block with lower frequency to finish producing data after they are expected by the next block with a higher frequency. The behavioral model then specifies how and when the circuit should transform data between its inputs and outputs. An example of a logic gate `AND` implemented in VHDL can be seen on the following code snippet:

```
entity AND_GATE is
port(
    a: in std_logic;
    b: in std_logic;
    y: out std_logic
);
end AND_GATE;

architecture behavioral of AND_GATE is
begin
    process(a, b)
    begin
        if ((a='1') and (b='1')) then
            y <= '1';
        else
            y <= '0';
        end if;
    end process;
end behavioral;
```

Listing 1: Implementation of AND gate in VHDL[8]

Although HDL languages can be used to design complex hardware blocks, a more popular approach to development of these designs is by the use of high level synthesis (HLS). This approach hints at the use of a higher abstraction level than hardware description languages do, such as C or C++ [12]. With this feature, hardware developers gain the opportunity to implement more complex designs much faster than by using HDL. The algorithm implemented in a high level language is first analyzed and verified for the selected architecture before its compiled into an RTL definition, which is later used in logic synthesis.

2.5 Logic synthesis

The process of conversion between the abstract definition of a logic circuit into the hardware implementation at a logic gate level is called logic synthesis [15]. The code written in a hardware description language including entity definitions and behavioral models are processed to verify and generate the required circuit logic.

The first phase of logic synthesis begins at the construction of a boolean network, which

[8] AND gate. Weijun Zhang. [online]. [cit. 2023-05-10]. Based on: http://esd.cs.ucr.edu/labs/tutorial/AND_gate.vhd

is defined by a set of boolean variables and their corresponding functions to determine the state of the output variable. The network is then optimized independently of the used technology, which includes simplification of the logic expression. The second stage of synthesis focuses on the specific target implementation, for which the simplified expression is further optimized. Finally, the synthesized design is properly analysed by calculating component and circuit delays and based on its results, the process of simplification and optimization can be redone to meet specific timing requirements and increasing circuit speed.

2.6 Debugging and profiling

To accurately determine the correctness of a designed circuit, there is a need for specialized hardware and software dedicated to debug and profile these implementations. Many FPGA manufacturers ship their products with debugging hardware modules built in and its corresponding software for ease of use. There are usually two stages when the debugging may occur based on specific platform:

- HDL simulation – This is the most commonly used technique for debugging created designs as it simulates the behavior of the circuit before it is loaded on the FPGA itself. One of the most known tools for this purpose is the ModelSim HDL simulator[9].
- Hardware signals and probes – Once the simulation process is done, the design can be executed on physical hardware as well to verify its functionality in real world usage. Some FPGAs include specialized hardware blocks, such as Xilinx Integrated Logic Analyzer (ILA)[10], which can connect multiple signals from the circuits within the FPGA and collect the data for viewing.

One of the most well known approaches is Waveform-based debugging, where the signal changes at specific time points can be viewed to confirm the requested behavior of the designed hardware block. The following figure displays connected signals and their behavior over a small period of time:

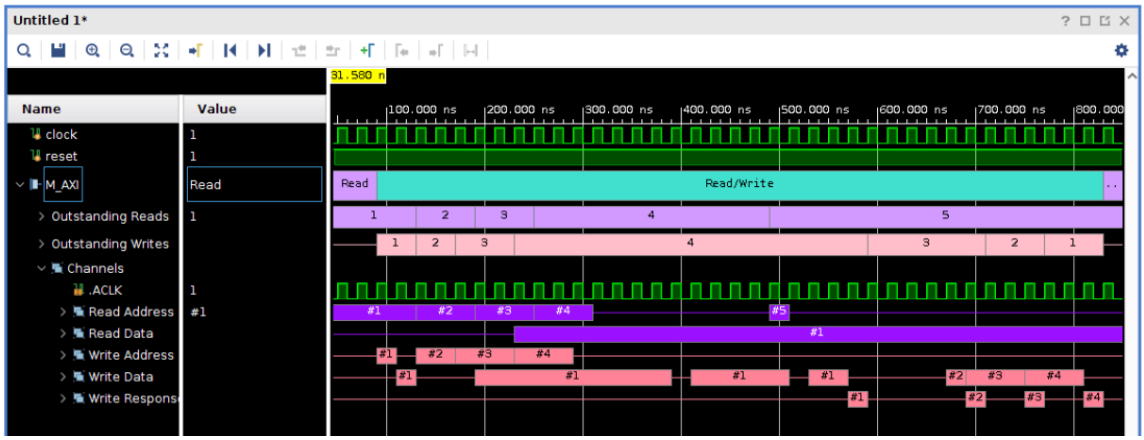

Figure 2.6: Waveform debugging example[11].

[9] https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user
[10] https://docs.xilinx.com/v/u/en-US/chipscope_ila

As already mentioned, FPGA development boards are produced with microcontrollers, which are often running in parallel with the FPGA and are used to control its execution. Software running on the CPU can be debugged with GNU Debugger (GDB), which is a well known tool to many software developers. The debugger can be connected through a specified connector available on the board, such as the Joint Test Action Group[12] (JTAG) interface. An integrated development environment (IDE) can then be used to load the synthesized circuit onto the FPGA and start the execution of the implemented software.

2.7 Microblaze

Microblaze is a soft-core processor that can be synthesized and placed on a FPGA board. It offers many configuration options that can be selected based on the task at hand. However, every additional feature uses additional FPGA resources that are limited based on the board model[13] and they should be carefully considered before they are enabled. Among these features are the data and instruction caches, interrupt and debugging support or hardware blocks for arithmetic operations[14]. External hardware blocks can also be connected to and controlled by the Microblaze via the advanced extensible interface (AXI).

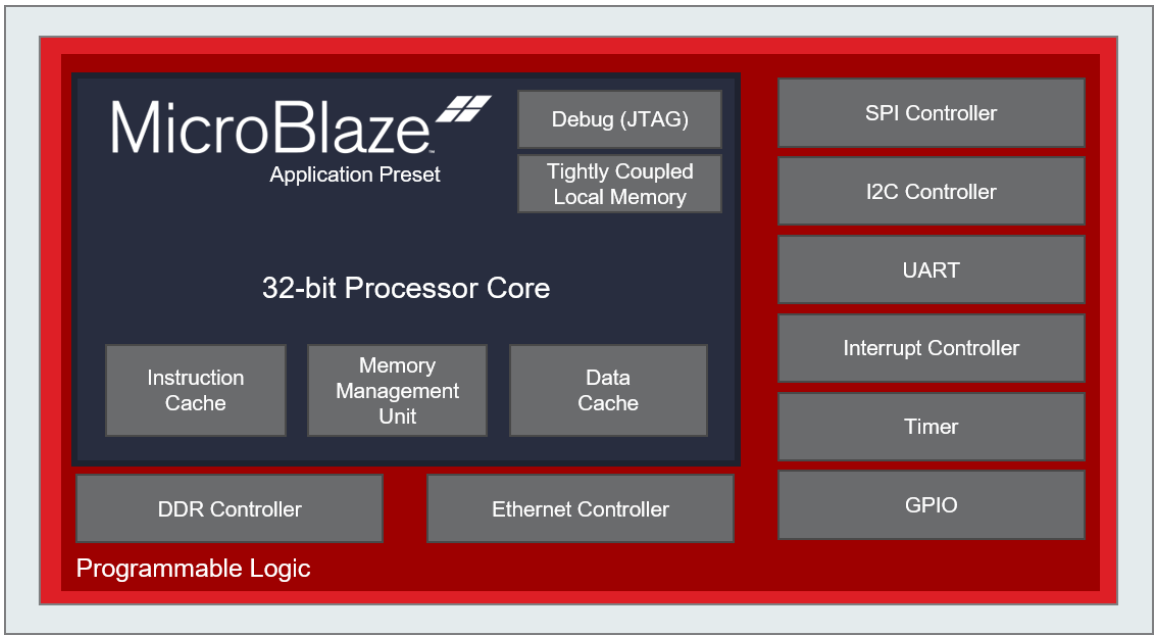

Figure 2.7: Microblaze architecture[15].

[11] AXI Basics 2 – Simulating AXI interfaces with the AXI Verification IP. Xilinx Inc. [online]. [cit. 2023-05-09]. Retrieved from: https://support.xilinx.com/s/article/1053935

[12] https://docs.xilinx.com/r/en-US/ug470_7Series_Config/JTAG-Interface

[13] https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

[14] https://docs.xilinx.com/v/u/2019.1-English/ug984-vivado-microblaze-ref

[15] Microblaze Soft Processor Core. Xilinx Inc. [online]. [cit. 2023-05-10]. Retrieved from: https://www.xilinx.com/products/design-tools/microblaze.html

2.8 Advanced extensible interface

The AXI is a communication protocol, which specifies multiple signals for reading and writing data between devices. Each operation is done in bursts allowing for multiple data transfers per request (not on AXI4-Lite). Similarly to I2C and SPI protocols, a participant in the communication can be either a master or a slave. Both communicating parties must perform a handshake before any data transmission can be initiated. The master sets its `VALID` signal once it is ready to transmit data and the slave sets its `READY` signal once it can obtain a request. Once both signals are set, the transmission is executed in the next cycle and both signals are unset.

Generally, the AXI protocol[16] defines 5 channels for specific requests, where each contains handshake signals for synchronization:

- Read address – The master requests a read operation from the slave based on given read address.
- Read data – The slave responds with data after successful handshake.
- Write address – Similarly to read operation, the master sets the write address for writing data.
- Write data – Requested data to write on the write address are sent to the slave.
- Write response – Response from the slave to indicate the status of the write operation.

The protocol above describes the AXI4-Lite interface specification, which is a simplified form of the original AXI4 protocol. Some additional signals, that are present in AXI4 include the data size, protection type, lock type for atomic operations or identifiers for multiple streams on a single channel.
In addition to AXI4 and AXI4-Lite, the AXI4-Stream protocol is also available for continuous streaming of data, which can be processed on demand. In AXI4, the burst must be completed before any operation can occur, while AXI4-Stream[17] allows for immediate data processing as soon as required amount of data are present. This approach can potentially reduce latency and is useful when large volumes of data (such as raw images) are transferred or the memory of the slave is limited.

2.9 Analog to digital converter

As the external signals are continuous and computers perform in a discrete environment, there is a need for a device that is capable of converting such signal into a form that can be digitally processed. The analog to digital converter (ADC) uses the process of periodical quantization to sample the external signal and provide its digital representation. The quality of an ADC is mainly determined by its sampling rate as it affects the reconstructed digital signal. Additionally, its resolution must be high enough to correctly assign a digital value to each input analog signal, otherwise the output data could be misinterpreted due to high quantization error [1].

[16] AMBA AXI and ACE Protocol Specification. ARM Limited. [online]. [cit. 2023-05-10]. Available at: https://developer.arm.com/documentation/ihi0022/e

[17] AMBA AXI-Stream Protocol Specification. ARM Limited. [online]. [cit. 2023-05-10]. Available at: https://developer.arm.com/documentation/ihi0051/b

Chapter 3

Cryptography

This chapter discovers different ways of ensuring data security by using cryptography algorithms based on asymmetric key pair. Additionally, their purpose along with their properties are also thoroughly compared between each other in order to find the best choice for usage in embedded systems. Some existing systems and their implementations of security solutions described below are also fundamental before moving on to design and implementation chapters of this thesis.

3.1 History

First usage of cryptography dates back to the Roman empire, when Caesars cipher [3] was created to „encrypt“ messages by shifting each character by a fixed number of positions down the alphabet to create a new message (see figure 3.1). However, this system is easily breakable even by hand and the original message can be recovered after finding out the number of shifts. More complex methods were later introduced for better encryption, but everything changed when computers first made their appearance. Modern technology brought many advantages which significantly changed our lives, however, the introduction of internet also came with many dangers. Suddenly, everyone who came online became vulnerable to many threats, including data theft leading to blackmail, forgery leading to loss of information and finances. It was vital to ensure data security by limiting access to it, making it unreadable for 3rd parties and proving their point of origin.

For this reason, there are multiple algorithms for different purposes, where each must have at least one of the following properties [7]:

- Authenticity – Identity of communicating parties and the origin of transmitted data is confirmed.
- Integrity – Data can not be modified during transmission between receiver and sender without detection.
- Confidentiality – Data can not be read by 3rd parties.
- Non-repudiation – Sender can not deny his intention of sending data at a later time.

The introduction of cryptography based on a randomly generated secret key had major impact on security today. It can be divided into two groups, where each group has different purpose, advantages and disadvantages.

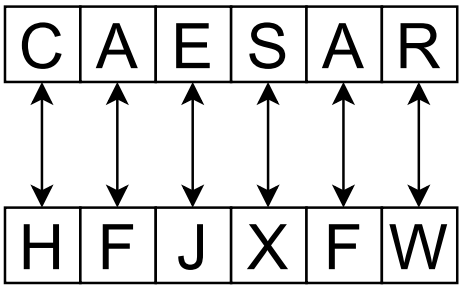

Figure 3.1: Caesar cipher shifted by 5 characters.

3.2 Symmetric cryptography

Symmetric algorithms use only one key, which has to be present on all communicating devices (see figure 3.2) in order to successfully encrypt and decrypt messages [6]. However, each party having the same key discards the properties of authenticity, integrity and non-repudiation, since its impossible to determine data origin and whether it has or has not been modified during transmission. The only property maintained is confidentiality, however, everyone in the group can decrypt and read the data, even when they are not addressed to them specifically. Leakage of the private key renders the key useless and it can not be used again, so the process of key generation and transmission needs to be redone.

Since symmetric key exchange is problematic over an insecure channel, it is important to ensure its delivery without it being compromised. One of the most common ways of accomplishing this is using the asymmetric key pair to encrypt the symmetric key and transmit it to other participants.

Alternatively, the Diffie-Hellman algorithm can be used between two communicating devices. The general idea described in article [9] is for participants A and B to select a private random key and derive its public counterpart, which is accessible to the other device. Afterwards, participant A can calculate the final key by combining its private key with the public key from B, whereas B computes the same value in similar fashion.

The biggest benefit of using symmetric key encryption is its performance. According to this study [5], one of the most popular block cipher algorithms AES (symmetric) records roughly 2x faster execution time on larger file sizes in comparison to the RSA algorithm (asymmetric).

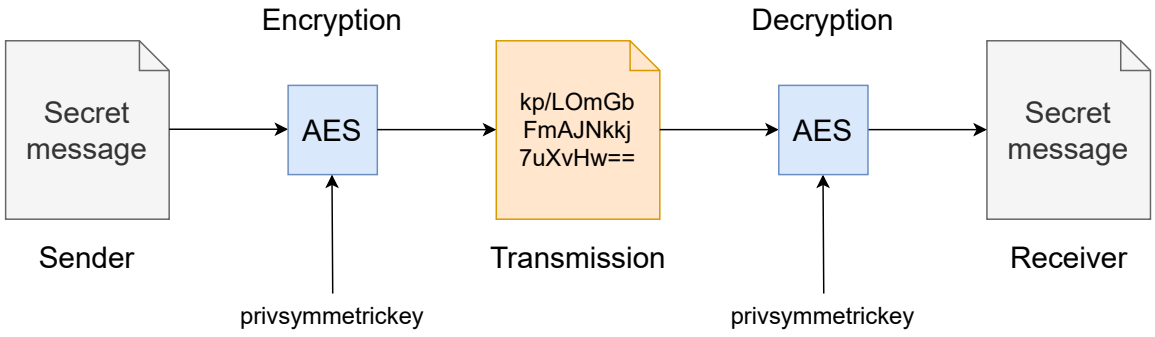

Figure 3.2: Symmetric encryption and decryption between two devices.

3.3 Asymmetric cryptography

In comparison with symmetric algorithms, asymmetric cryptography uses two keys (see figure 3.3), also known as a key pair, where one of them is private and the second one, derived from the private key is public [6]. This approach allows for better security as the private key never leaves the device where it was created, however, the performance due to the need of two keys is degraded as has been shown in previous section. The encryption process ensures confidentiality as it uses the receivers public key to encrypt the message, which means that only the receiver can successfully decrypt the data as it owns the private key.

In addition to performance disadvantage, memory usage is also quite different between algorithms. According to the same study [5], the RSA algorithm used roughly 3x more memory than Blowfish symmetric cipher. These differences should be mostly taken into account when the used computation device has insufficient resources, such as a microcontroller.

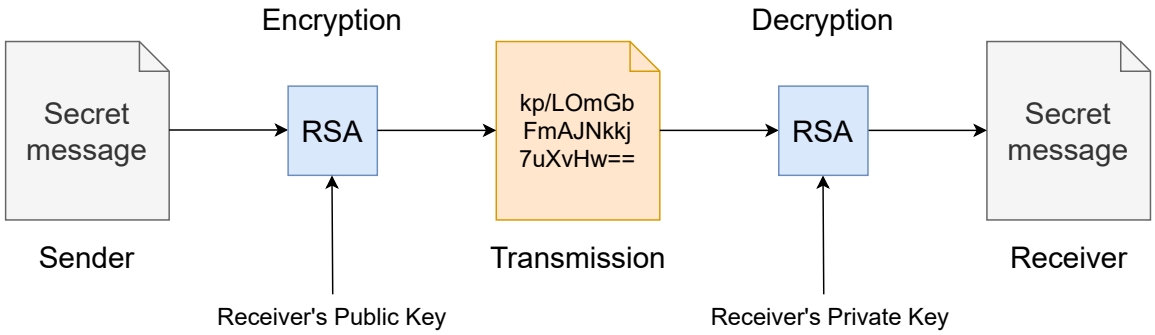

Figure 3.3: Asymmetric encryption and decryption between two devices.

3.4 Signature Algorithms

In addition to data encryption, asymmetric algorithms can also be used for generating digital signatures to ensure authenticity, integrity and non-repudiation [6]. A digital signature is a unique imprint of data that originated on the device, which executed the process of signing. In this process, the private key is used to create a signature, which is then sent along with the data to the recipient. After the transmission is done, the public key can be used to verify the signature and ensure that it has not been modified by a 3rd party or generated on a different device.

However, large messages such as files can require a lot of processing power for generating a signature, so more often than not, a message digest of the message is created and used for signing instead of source data themselves. After the signing process is done, the original message (or its encrypted version for ensuring confidentiality) is sent along with the signature to the receiver, where the same hashing algorithm is used to generate the hash of the message again. Hashing algorithms are generally very fast, especially when compared to signature algorithms, which is why this approach is widely used. A signature generation process with hashing included is shown in the following figure.

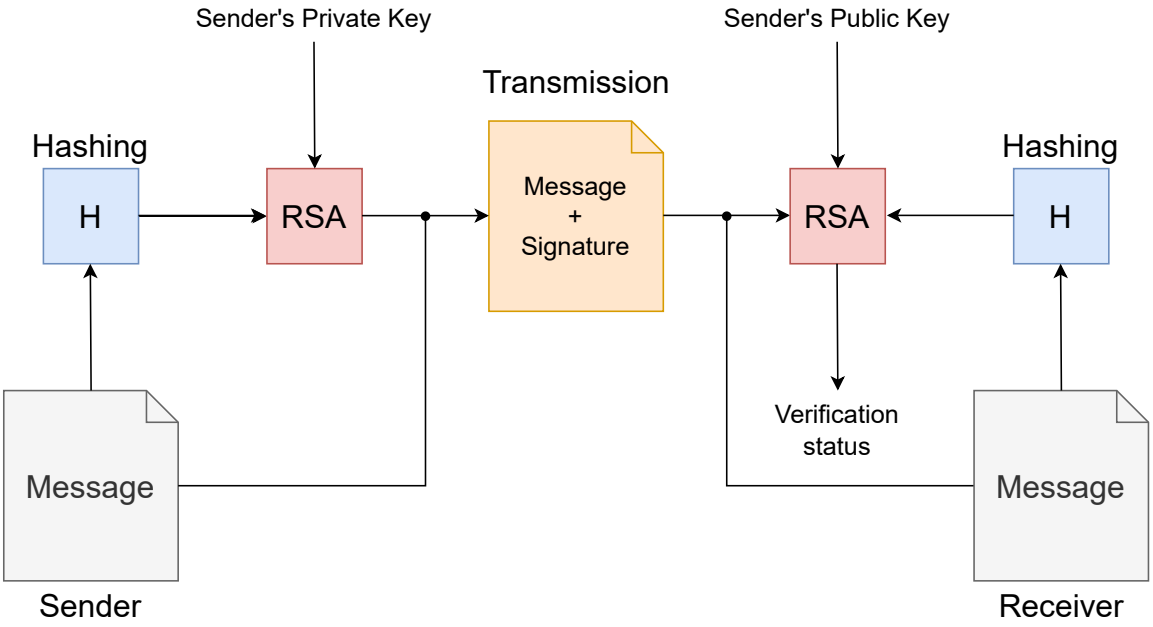

Figure 3.4: Signature generation and verification process with hashing.

3.5 Rivest-Shamir-Adleman (RSA)

The RSA algorithm is one of the oldest asymmetric cryptographic algorithms today. It is based on the problem of integer factorization, which relies on computational difficulty of factoring the product of two large prime numbers [6]. The product is also known as the modulus, which is used in the process of encryption and decryption [21]. Deducing the factors of this modulus would lead to finding the decryption key, however, this is a computationally intractable operation if the key is large enough. The usage of a large key leads to slow performance and high memory usage, which is why its becoming obsolete and being replaced by newer algorithms.
In terms of data encryption, the public key is used to encrypt a message and can only be decrypted by the owner of the private key, where both operations are conducted by using modular exponentiation. The smallest key size that is considered secure today is 2048 bits according to NIST recommendations [17].

3.6 Digital Signature Algorithm (DSA)

In asymmetric cryptography, it is vital for the algorithm to be efficient while ensuring the secrecy of the private key. The Digital Signature Algorithm (DSA) is mainly used for signature generation and verification by using modular exponentiation, which is considered as a simple mathematical operation. However, the inverse discrete logarithm problem is computationally intractable similarly to the integer factorization problem of the RSA. However, according to study [18], its security strongly relies on using a random nonce during signature generation and omitting this step could lead a potential attacker to recovering the private key.

The similarities with RSA continue with regards to the private key, which is also recommended by NIST [17] to be of at least 2048 bits in size to remain secure enough for

current standards. However, a better alternative would be the elliptic curve variant of the DSA algorithm also known as ECDSA, which has become more and more popular in the last decade. The reason for its popularity is due to the usage of a much smaller private key while maintaining the same security level. A 256 bit key is equivalent to a 3072 bit RSA or DSA key according to NIST [17], which implies improvements in terms of reducing key storage requirements.

3.7 Message Digest

A message digest (or a hash) is a unique value, which represents the data it originated from. The same message always yields the same hash of fixed size and the original message is very hard to restore from the hash itself [6]. Its size is usually in the range of tens of bytes based on used algorithm (eg. SHA256 uses fixed size of 32 byte message digests), which in most cases is much less than the original message. Its size also affects the probability of finding the same hash for a different message. It should be computationally infeasible to find two distinct messages that hash to the same message digest. Some message digests such as the MD5 algorithm have become obsolete since they have been cryptographically broken and are no longer suitable for safe usage[1].

As shown in the next figure, a difference in a single character can change the entire message digest, whereas the difference in message length does not affect its size.

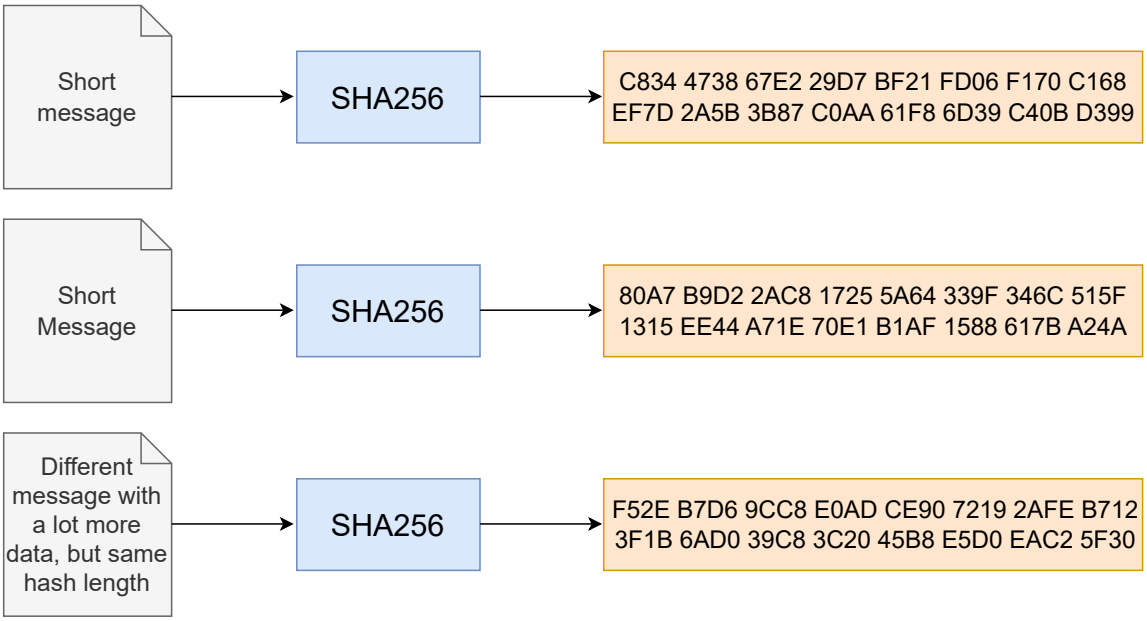

Figure 3.5: SHA256 algorithm outputs for different input messages.

3.8 Random Number Generator

As mentioned above, some cryptographic algorithms strongly rely on randomness in order to be practically usable in real world applications as shown in study [18]. The DSA algorithm

[1] MD5 vulnerable to collision attacks. Chad R. Dougherty. [online]. [cit. 2023-05-10]. Available at: https://www.kb.cert.org/vuls/id/836068/

requires a random nonce for each signing instance, otherwise the used private key could be retrieved and all created signatures would be impossible to validate. For this purpose it is necessary for a random number generator (RNG) to be present during the process of signature generation. However, it is practically impossible to create a true RNG on current hardware as any implementation must have a predefined algorithm. For this reason, a more appropriate term to be used in relation to computers would be a pseudo-random number generator (PRNG) or a deterministic random number generator (DRNG).

A PRNG usually consists of two main parts, which together produce a (pseudo) random value on its output (see figure 3.6):

- Internal state – Represents the current state of the generator. Generally, it is a random value that has been created in previous generation process with the exception of the initial state, which is expected to be set from an outside source before first usage.
- One-way function – The algorithm used to generate the random value, which will also become its next internal state. It should be computationally infeasible to compute its inverse function.

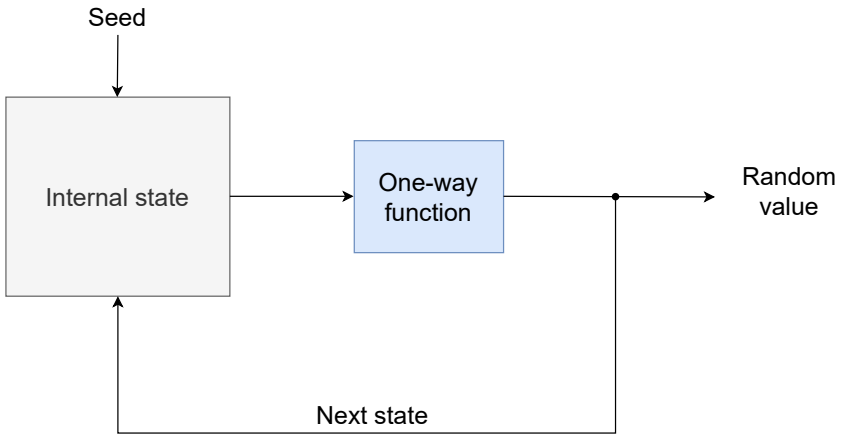

Figure 3.6: Pseudorandom number generator structure.

One of the oldest and most known implementations of a PRNG is the linear congruential generator (LCG). Its one-way function as shown in [7] is defined by a simple recurrence relation $X_{n+1} = (aX_n + c) \mod m$, where X_n is the current internal state of the generator, a is a multiplier, c is an increment and m is modulus. The specific combination of parameters define the generators statistical properties, such as period. Period is a number of random values in a sequence that the generator yields before the same sequence starts repeating. In case of usage in ECDSA, it is vital for the generator to have a long period since repeated nonces would have destructive consequences.

According to [7], a generator should be cryptographically secure (also known as CSPRNG) in order to be safely used in asymmetric algorithms by meeting following requirements:

- For an already generated sequence of n bits, the probability of predicting the $n+1$th bit is non-negligibly larger than 50% by a polynomial-time algorithm.
- If at least a part of the internal state is revealed, it is not possible to reveal any states generated in the past. Additionally, the knowledge of the entropy (see next section) during execution should not allow for finding out future states of a CSPRNG.

3.9 Entropy

Before the generation process of a PRNG begins, its internal state must be initialized with a value also known as seed [7]. Without the seed value, the PRNG would generate the same sequence of values each time it was instantiated since the algorithm stays unchanged. The generator can also be reseeded during its operation in order to prevent its current period from ending. The value used for seeding the random number generator must be random itself, which is again a challenge in a deterministic execution environment. For this purpose, external sources of noise are often considered as good options for generating random data, which are also known as entropy sources. Modern hardware has means for extracting entropy with the use of device drivers and peripherals such as a mouse or a keyboard and capturing their events into the entropy pool[2]. Another option is the use of a sensor which collects data from the environment, which are later converted into a digital form. Its usual to use multiple external sources or to do multiple readings and combine their outputs to generate a stronger entropy. Strong entropy makes it very hard for a potential attacker to determine the random stream of a PRNG, which in turn ensures better security during signature generation by creating unpredictable and unique nonces.

3.10 Merkle tree

The Merkle tree is a special form of a cryptographic structure usually in form of a binary tree designed by Ralph C. Merkle [16], which is used to store message digests (see figure 3.7). A binary tree is a structure, where each node can branch into at most two child nodes and become a parent node. Nodes that do not have any child nodes are known as leaves and the first node in the structure is the root node. The Merkle tree is usually used as an optimization to improve performance by reducing the amount of data that need to be cryptographically signed. Hashes of input data are stored in leaf nodes of the tree until it becomes full or until a root hash is requested. Each pair of child nodes is hashed together and the result is stored in the parent node. This process is done recursively for all child nodes until the root node is reached. The root hash is the only hash used in a signature generation process, which can greatly improve performance depending on the size of the tree. This approach uses additional resources for storing or regenerating hashes for later verification, but its usually an accepted compromise if the performance is restricted on some devices and a signature can not be created for each message (hash).

[2] https://linux.die.net/man/4/random

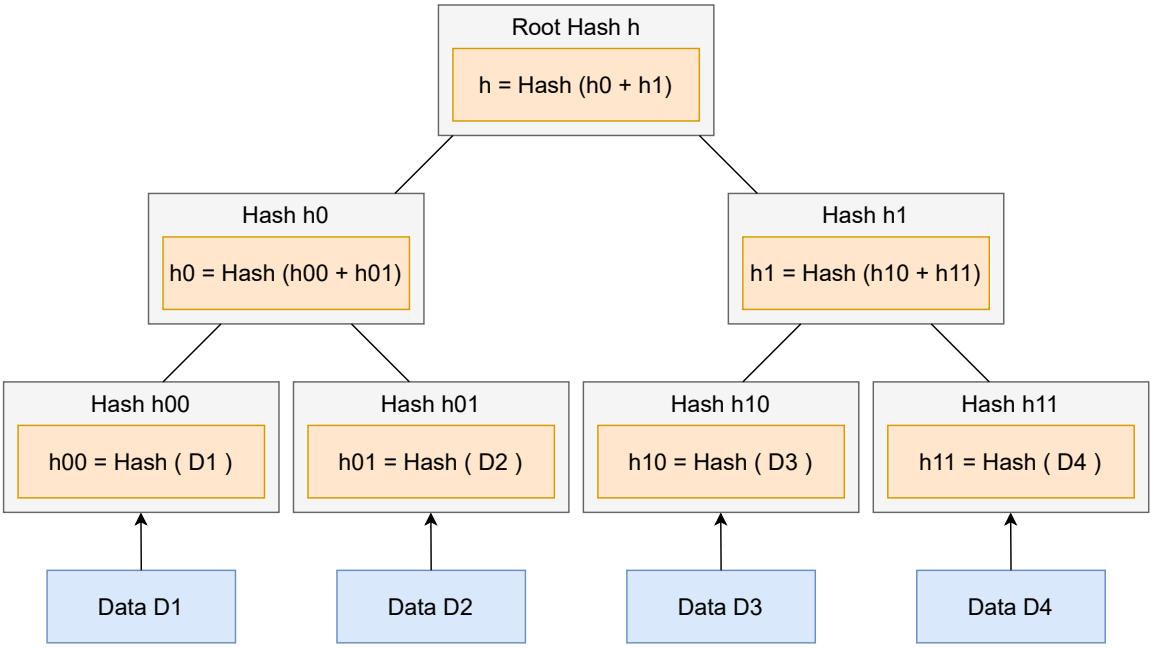

Figure 3.7: Merkle tree with 7 nodes for storing 4 hashes.

3.11 Hardware acceleration

As embedded systems are usually restrained in terms of resources, the computation complexity of cryptographic algorithms become more noticeable on such systems. Nowadays, many new microcontrollers come with hardware blocks specifically designed to increase the devices performance in cryptographic algorithms by offloading the task from their CPUs. This affects the overall responsiveness of the entire system as it is able to focus on different tasks at hand, that can not be accelerated otherwise. The execution time is usually also greatly reduced in comparison to the usual CPU. The data along with the job description are written to the hardware blocks registers before the execution starts and the results are usually signalised by interrupts. A study on embedded systems hardware acceleration for cryptographic algorithms [13] shows that performance gains on RSA encryption and decryption can range from 26x to 57x depending on key size and a significant improvement can be seen on hashing algorithms from SHA160 (6x) to SHA512 (278x).

Chapter 4

Design

As mentioned earlier, this thesis is a part of a larger research project `SECUSEN`, which focuses on ensuring data authenticity and integrity by using cryptographic signatures and timestamps. The idea is to generate a signature for given data as close to their source as possible to diminish the amount of weak points that could be taken advantage of by potential attackers.

The project uses an FPGA camera sensor as a source of data that are required to be timestamped and signed in a way that their point of origin would be undeniable. Since the camera sensor is implemented on a resource constrained device with limited computing potential and memory capacity, the choice of the correct cryptographic algorithm becomes crucial.

The focus of this thesis is to gather message digests of the image data created on the camera and cryptographically sign them on a FPGA. The key part is hardware acceleration of the asymmetric algorithm to ensure that the data collection process will not be limited by inadequate performance. Additionally, a random number generator along with entropy collection algorithm will also be implemented within the FPGA to ensure private key confidentiality. Finally, the resulting hash with the generated signature shall be transmitted to the Linux subsystem through a predefined communication interface.

For the purpose of design, implementation and profiling, the Zynq ZC702 evaluation board will be used as a platform for this thesis as it provides the interface for debugging as well as the Linux operating system for communication with a remote server for collected image data and their signatures. The evaluation phase includes the measurement of execution time, memory usage and verification of the generated signatures.

This chapter will focus on design choices based on aforementioned requirements and limitations, from hardware design that includes intellectual property (IP) selection and their individual configuration possibilities, to software design which includes cryptographic algorithm selection, implementation and profiling proposal, possible ways of optimization and hardware acceleration.

4.1 Cryptographic algorithm and hashing

One of the most important parts of the design is choice of the asymmetric signing algorithm that will be implemented on FPGA. Rivest Shamir Adleman (RSA) is one of the most known and used algorithm for signatures and in many cases, its performance is adequate and can be run in real time for large amounts of data. However, the usage of large keys, which nowadays have to be at least 2048 bits in size to be considered secure, brings complications in terms of memory requirements. For this reason, the Digital Signature Algorithm (DSA), specifically its elliptic curve version (ECDSA) is a better fit for our needs as the same security level can be achieved with much smaller key size of 256 bits, which in turn consumes a lot less memory.

There are many existing implementations of ECDSA in 3rd party libraries that are likely to be well optimized, tested and verified. The MbedTLS library focuses on implementation of cryptographic functions on embedded systems, which means that many ARM architectures are supported with performance and code size in mind[1]. The Microblaze softcore processor on FPGA is also officially supported by MbedTLS, which makes it a strong option for our requirements. However, the choice of elliptic curves that can be used in signature generation and verification processes is not too broad, as some elliptic curves are either too small for them to be considered safe or unnecessarily large, which would lead to performance degradation. The list of 256 bit Montgomery curves available on MbedTLS include the `BP256R1`, `SECP256K1` and `SECP256R1`, all of which need to be evaluated to make the final decision.

An interesting alternative worth mentioning would be `Curve25519`, which promises better security and better performance than any of the aforementioned elliptic curves [8], however, during the time of implementation of this thesis, this curve is not yet fully supported in MbedTLS for usage in signature generation and verification.

As the volume of data output is expected to be relatively large, it is almost necessary to use a hashing algorithm in order to be capable of generating and verifying signatures in real time. MbedTLS supports MD5 and multiple versions of SHA, from SHA-1 (nowadays considered outdated and not secure enough along with MD5) all the way to SHA512. For the same reason the 256 bit keys were selected for ECDSA, the hashing algorithm of choice is SHA256 as it is secure enough, but due to resource restrictions any larger hash could have a considerable performance impact.

4.2 Random number generator

The revelation of our private key to a third party would have destructive effect as every signature created from that point would be untrustworthy and their successful verification would yield no proof of authenticity or non repudiation. During signature generation process, it is vital for the key to be secured by creating a random nonce for each input message. By omitting or repeating the random nonce the private key could be recoverable from the generated signature and original data. For this reason, the signature generation algorithm needs to include a random number generator with good statistical properties, with as long period as possible and hard to predict future values. Some well known random number generators include the Mersenne twister or Arc4Random, however the former uses a large amount of memory and the latter is relatively time consuming [19]. As our software

[1] https://github.com/Mbed-TLS/mbedtls

implementation focuses on small code size and performance due to memory restrictions and limited processing power, the proposed generator for usage in this thesis is Permuted Congruential Generator (PCG). It has been shown to have very good statistical properties while being compact and fast to generate the next state (value).

4.3 Entropy collection

Every random number generator needs to be initialized before usage by seeding it with random entropy. All algorithms are by design deterministic, and randomness is quite complicated to implement in such environment. Random number generators in fact are not random at all, as their next state is always generated from current state by predefined sets of calculations, which are as already mentioned, deterministic. This is why they are required to have as long period as possible, otherwise the same random values would be generated too often. The only way to ensure true randomness is to seed the generator with values gathered from sources that can not be predicted and regularly reseed the generator before its period ends. Some of those sources include sensors that gather data from the environment, such as temperature or humidity sensors. The data gathering process is analog, where the collected value is represented by the amount of voltage on the sensors output. The analog representation is then processed by analog to digital converter (ADC), which transforms the signal into a format that is understood by computers and can be further processed in software. It is good practice to combine outputs of multiple sensors or to count the occurrence of one or zero bits and run the gathering process multiple times to generate even stronger entropy. The implementation of an ADC IP already exists on the FPGA and it will be used to gather entropy from a temperature sensor in order to properly seed the PCG PRNG prior to creating the first signature.

4.4 Computational unit in the FPGA

One of the most known processors that can run software on FPGA is Microblaze. It can be synthesized in 32 or 64 bit architecture, with various features enabled or disabled based on what algorithm and how much data it is meant to process. All relevant features need to be thoroughly tested and combined to reach its full computing potential. It should be noted that most features require additional resources on the FPGA to be used, which could be a potential complication if its already highly utilized.

The most computing power is expected to be consumed by ECDSA, specifically mathematical operations executed on curve points, which are in our case 256 bits in size. On a 32-bit soft-core processor, these calculations need to be divided into smaller operations and executed sequentially. The Microblaze configuration wizard allows for adding an integer multiplier and divider, which should provide better results than its built-in version. Some operations in MbedTLS are implemented with the use of bit shifting, so the barrel shifter option could also produce some positive impact on performance. On the contrary, the floating point unit is not expected to have any impact at all as the algorithm only processes large integers and would not utilize this hardware block, so it should remain disabled. Some features that may or may not cause any change in execution time are data and instruction caches, as the block RAM is usually fast enough to provide required data, however, the result will be known during evaluation.

As already mentioned, memory resources on the Zynq ZC702 FPGA are also heavily

limited. This fact dramatically reduces the set of options for third party cryptography libraries, as most of them require at least hundreds of kilobytes, which is not ideal for our use case. The Microblaze processor can be configured with up to 128 Kilobytes of block RAM with a single BRAM controller, which is shared for both program and data. It is possible in theory to configure multiple BRAM controllers allowing for more memory blocks, but this approach is better avoided as it complicates the design and the focus of this thesis is low memory usage and high performance.

It is well known that the higher frequency the processor supports, the better are its computation capabilities. The Zynq IP will provide its clock source connection to all other IPs within the block design, including the Microblaze. It can be configured to run at a specific frequency, which is naturally crucial for achieving fast execution time.

4.5 Application evaluation proposal

In order to properly determine the ECDSA signature generation complexity, it is necessary to modify the board design by adding an AXI timer for measuring execution time of the implemented solution. Every IP within the board package has its drivers exported along with them for easy initialization and manipulation. To measure time duration of a specific code segment, the only required action is clearing the timers state and starting it before the segment begins and stopping it after the segment ends. All functions are implemented in the AXI timer IP driver, which take the timer context and ID as parameters. In debug build type, breakpoints can be used to stop execution at a certain point and read the value of the timers register. However, in release build type, the debugger has no ability to stop at breakpoints, which means that another solution is needed. If the local memory of Microblaze has unused space, we can write into the memory at a specific address by using input/output functions from within the board support package (BSP). This address can be inspected during execution of the application by memory monitor, which allows for reading data at specific memory addresses without the debugger being attached.

This approach allows for efficient evaluation of different Microblaze configurations, with different clock frequencies on its input. However, it would be extremely time consuming to determine which specific instruction or function is the processor spending most of its time executing. The simplest way of finding the most time consuming code segment is by the use of profiling. The `gprof` is an intrusive profiler, which requires a timer IP, interrupt connection and additional modifications in the BSP and the application itself in order to run and generate an informative output. The TCF profiler can be executed in debug build type on Microblaze and is non-intrusive. It only requires for the debugger to be enabled in the board design, but does not need additional timers or software modifications in order to run. The main reason to use the TCF profiler is due to the support of stack tracing, which can be useful during code inspection.

The application itself can be built with multiple optimization choices, which focus either on code size or execution performance. The `-Os` compiler flag optimizes the application for the smallest code size possible, but the algorithm may be slower in comparison with the `-O3` flag for best performance, which in turn can be too large to fit into the small memory block. All combinations should be thoroughly tested and evaluated to determine which optimization mode fits our needs the most by having the least compromises.

4.6 Xilinx Vivado HLS

The Vivado Design Suite comes with support for high level synthesis (HLS), which creates an opportunity to design a custom hardware block based on a code segment. During the evaluation stage of our solution implementation, the profiler is used to determine the segment with longest execution time. The function is then inspected and the possibility of its implementation within HLS determined as there are limits on C constructs, which can not be used in high level synthesis such as dynamic allocation, recursion or system calls[2].

Once the code segment is implemented in HLS, its behavior needs to be tested to ensure it produces correct output. Vivado HLS supports creating test benches to compare its results with expected output. Before synthesis begins, the communication interface can be modified based on the function parameters. Many IPs in the Vivado Design Suite support the AXI interface, including the block RAM, ADC or timer, all of which are expected to be connected through the AXI interconnect. For this reason, it is strongly recommended to use this interface for the custom IP as well, though not required. Afterwards, the synthesis is executed and the usage of FPGA resources with timing checked to ensure it will fit on the board and does not have too high latency. The co-simulation feature of Vivado HLS is then used to verify the functionality of the newly synthesized IP before it is exported along with its driver. Afterwards, the IP can be imported into Vivado Design Studio and added into the block diagram, preferably connected to the Microblaze through aforementioned AXI interconnect. The exported driver is compiled within the BSP and can be used in the same fashion as other AXI enabled hardware blocks.

4.7 Communication interface

Before beginning, it is important to note that this part has been designed for project `SECUSEN` and it will not be implemented in the final solution of this thesis. Nonetheless, a detailed description of the proposed communication interface will be provided anyway.

As the network communication must be implemented on Linux side, there is a need for an interface between the FPGA and the Linux subsystem. The most usual way of data transmission between the two sides is through registers that create a memory block. The BRAM interface can be synthesized on the FPGA and its start and end addresses can be specifically set within the Vivado Design Studio. The specific system of transmission and communication is determined by the programmer. Usually, the first register in the memory is used as a control register. Its size is 4 bytes, which means it allows for setting 32 bits that can be represented as flags. Depending on the data volume that needs to be transmitted, next n registers can be reserved for the data itself. ECDSA signatures are 64 bytes in size, which requires 17 registers in total (1 control and 16 data registers). Flags in the control register then describe the state in which the data registers are, such as ready or busy. The other communicating party must poll the control register and wait until a combination of specific flags represent a valid state to perform an action. When the `busy` flag is unset, it means that no operation is being performed and one of the communicating parties can take ownership of the reserved memory block. Once a signature is created on the FPGA and the `busy` flag is unset, the process of data transmission can begin. The program on FPGA sets the `busy` bit and writes the signature into data registers. The Linux subsystem is then signalled about new transmission by setting the `ready` flag and unsetting the `busy` flag.

[2] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Unsupported-C/C-Constructs

Due to the control register being polled, the other side decodes the state of the interface as ready and sets the `busy` flag on to avoid concurrent access. At last, it reads 16 data registers containing the signature and unsets both `ready` and `busy` flags, signaling that the read operation is done. More complex structures like queues or stacks are also possible with more control flags and multiple data blocks in case the reading side becomes busy, resulting in either data loss or waiting and slowing down the entire system. In case of our implementation, the FPGA is expected to be slower than the Cortex A9, so it should be capable of reading data faster than the time it takes to generate another signature. It should be noted that this communication design is unidirectional and a secondary structure needs to be created if a bidirectional communication is necessary.

Chapter 5

Implementation

Last chapter described how each part of the implementation phase should be executed in order to create a working, secure solution. However, the final product has some differences from the original design as some specific decisions could not have been made earlier. Additionally, there have been some challenges that needed to be resolved in order to achieve the expected result. These difficulties will be described along with why they were encountered and how they were resolved in this chapter. Mainly, the focus remains on the changes and additions to the original design. Detailed evaluation results will not be discussed in this chapter, although it had been done in parallel to decide the best course of action to finalize the solution.

5.1 Hardware and software preparation

The Zynq ZC702 evaluation board supports multiple boot modes which can be selected by the user. The SW16 switches on the board were all set to 0 to enable JTAG debugging and disable loading from the SPI or the SD card. There have been some complications with the debugger in Vivado Design Studio with the Linux operating system running. The debugger could not connect to the board correctly and load the compiled software. Since its execution was not necessary during the implementation and evaluation stages, the Linux subsystem remained offline to avoid this issue and any other potential problems.

On the software side, the Vivado Design Studio needed to be installed with the Vitis IDE, Zynq 7000 series drivers and board packages for development. During different stages of implementation, it was necessary to switch to different build versions as some bugs and issues were encountered. Finally, the most suitable version (although with some issues as well) was 2020.1 and thus it had been used in this thesis. The host operating system used was Windows 11.

5.2 Board design

First of all, the performance of the ECDSA implementation in MbedTLS needs to be evaluated with different features enabled or disabled to find the best performing configuration with the least FPGA die usage. Beginning with the input clock frequency, it starts at 50 MHz on the Zynq IP output clock pin `FCLK_CLK0` by default. Doubling the frequency to 100 MHz keeps the integrated circuit stable without any detected delays, which allows us to try a higher frequency. Raising the frequency to 166 MHz should provide more than 3x

the performance than its original value, but the circuit can no longer handle such high frequency without losing stability. The highest possible value found with stable performance is roughly 125 MHz at the Microblaze clock input. This allows us to generate the signature approximately 2.5x faster, which is directly proportional to the increase in clock speed and this fact is confirmed after running performance tests with the AXI timer (see table 6.1). Further additions of the 32 bit multiplier, barrel shifter and branch target register cache further increased the speed by 6.7x. Other features available for Microblaze did not bring any improvements in terms of performance for our use case and thus remained disabled to keep the die size as small as possible (see table 6.2).

The AXI Timer, XADC and block RAM with BRAM controller are also available by default in the Vivado Design Suite. The analog to digital converter for entropy collection was configured to use AXI as its interface and only a single channel being the temperature sensor. All ADC alarms were disabled, otherwise the design could not be synthesized without connecting their pins. The block RAM was configured to be of size 4096 bytes, which can allow for additional data to be transferred along with the signature, such as timestamps or hashes. Like all previous peripherals, it was also configured to use the AXI interface, which makes all of them available through the AXI interconnect to the Microblaze processor. Every other connection has been done automatically by the Vivado Design Suite, which generated an additional processor system reset IP and connected all IP clocks to the Zynq IP clock output `FCLK_CLK0`.

At last, the HDL wrapper was generated for the design, its synthesis was done, the bit stream was generated and the output exported. The platform project for creating applications could then be created in the Vitis IDE, which used the exported design along with the bit stream to properly configure the board support package (BSP).

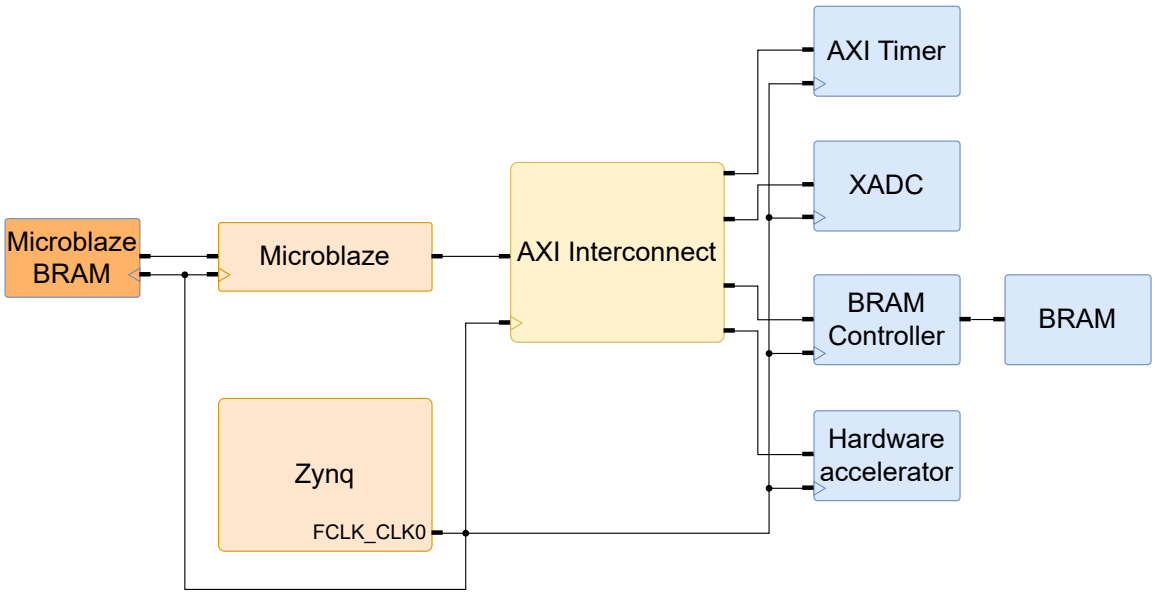

Figure 5.1: Simplified board design in Vivado Design Suite.

5.3 ECDSA and SHA256 integration

With the Microblaze platform project we are able to create a new application based on this platform and integrate the MbedTLS library to use its functions for ECDSA and SHA256. First of all, we create an empty C application and add a new file for the main function. To maintain the smallest possible size of the library, we can use the `mbedtls_config.h` file to enable necessary or disable unneeded features. The necessary features of MbedTLS include:

- `SECP256K1`, `SECP256R1` or `BP256R1` – The used curve for signing (only one is required at a time).
- `BIGNUM` module – All arithmetic and logical operations on big numbers by using decomposition into smaller data types are implemented in this module.
- `ECDSA` – Functions necessary for the ECDSA algorithm.
- `ECP` – Implementation of optimized algorithms for specific curves.
- `SHA256` – Implemented message digest.

The `check_config.h` file must be edited to not require SHA224 alongside SHA256 and fix the corresponding compilation error. Furthermore, the ASN encoding and decoding functions are by design required for ECDSA functions to be available, so its important to modify the `check_config.h` file and remove all functions that use ASN encoding or decoding. The signature will therefore be passed in its raw format as an (r, s) tuple, totalling 64 bytes. This format is not human readable and may contain unprintable characters, which is unsuitable for saving into text files. This may require additional encoding before sending it to a remote database, but it will decrease the total library size. In terms of performance, the removal of encoding does not produce a noticeable change.

Once the feature selection is done, the library is built and is prepared for use in our application. The algorithm requires a private key, which can be generated either by MbedTLS or other generally known tools and libraries such as OpenSSL. The key pair is generated outside of Microblaze as it is a one time operation only, however the application can be modified to generate a new private key every time the FPGA boots and provide the public key through the block RAM before any signing operations occur. As most features are disabled within MbedTLS, it is important to have the key in its binary format and save its 32 byte value into the block RAM in order to successfully load it with the library. Afterwards, we need to create and initialize structures that will hold the private key and pass it to the algorithm once it is needed. The EC key pair structure holds information about the stored key, including its curve type and public key if it is provided. During evaluation, three curves were tested to determine the best performing one. The best option for our use case is the `SECP256K1` curve as the `SECP256R1` and `BP256R1` did not perform as well and would not be able to be used in real time (see table 6.3). The MbedTLS structures for storing the (r, s) signature were also initialized as the ASN encoder and decoder has been disabled.

At this stage, the algorithm is almost ready to be run as everything is prepared except for the random number generator. For evaluation purposes of the ECDSA algorithm itself, a dummy function was implemented to fill the provided buffer by MbedTLS with static data. This function would later be replaced by a real random number generator

once evaluation is done in order to get correct profiling results (see table 6.7). The dummy function is then passed along with other parameters and a hash to the ECDSA signature (and verification) function.

5.4 Entropy source and RNG implementation

Until now, the ECDSA algorithm did not use a fully secure solution to generate new signatures as the Microblaze has been missing randomness. This was only acceptable for determining its performance, but can not be used in real world applications. In previous chapter, the PCG random number generator was selected as a fast, small and statistically secure solution. This decision has changed as a study has shown [20] that the PCG is not cryptographically secure. As a replacement to the PCG generator, the ChaCha20 PRNG has been selected for its cryptographic security [4]. The algorithm implementation by Stephen Mueller[1] supports automatic reseeding once 5 minutes have passed or 2^{30} bytes have been generated, which requires a time source to be available. The AXI timer present in the board design can be used for this purpose, so it is passed to the initialization function of the generator.

Missing randomness on the Microblaze was a problem that has been relatively simple to resolve. The ADC IP that has been configured in the previous section can be accessed by the `SysMon` functions in the BSP. After initialization of the ADC is complete, its 16 bit value can be obtained. To increase the strength of the entropy, multiple readings are done with a `SysMon` reset in between them to avoid getting unchanged values. The occurrence of 1 bit values within the 16 bits was then counted and based on a defined threshold, the output entropy bit was set accordingly. The ChaCha20 PRNG can be seeded with variable sized buffer filled with entropy data and for the purpose of this thesis a total of 64 readings (8x8 bits) are done on the ADC. Once the entropy is collected, the generator is seeded and its structure along with the function are passed as parameters to ECDSA to be used in blinding of the private key by generating a nonce.

5.5 Optimization with the Merkle tree

As the implementation progresses and more configurations and conditions are evaluated, it is possible to roughly calculate the average framerate we can expect to get from the FPGA camera. According to table 6.2, the execution time for generating a single signature is 88 milliseconds in the best case, which means that the Microblaze processor is capable of signing around 11 frames per second (FPS), which is relatively low for current standards. To increase its computing capabilities, it is possible to reduce the amount of signatures per second by using the Merkle tree optimization. The idea is to generate a SHA256 message digest and store it within the tree until all leaf nodes are used. Afterwards, each pair of child nodes is recursively hashed together into its parent node until the root node is reached. The root node is the one hash that has to be signed, whereas other hashes simply need to be stored in a database in order to successfully verify the data. If one of the hashes in leaf nodes is changed, the root hash changes as well and the signature can not be verified as the integrity has been breached. Hashing algorithm is generally known to be faster than any asymmetric cryptographic algorithm as can be seen in evaluation chapter (see table 6.5). The memory requirements for storing hashes and the algorithm code itself are low even for

[1] https://github.com/smuellerDD/chacha20_drng

a memory restricted environment such as the block RAM in the FPGA, which is important for our use case. The only disadvantage of this approach is the need to either use more storage for computed parent node hashes or to compute those hashes again later in order to verify the signature. However, verification process is most likely to be done on a much more powerful machine than the FPGA, so the advantages of this optimization outweigh the disadvantages.

5.6 Code profiling

The aforementioned Merkle tree is objectively a good and quick to implement workaround for solving inadequate performance of the ECDSA signing algorithm in Microblaze. However, the Vivado Design Studio offers features that could potentially bring performance or memory usage improvements. Generally, most algorithms have specific code segments that require a lot of processor cycles and take longer execution time than the rest of the algorithm. Often it is enough to simply deduce based on experience, which part could or could not be computationally demanding. However, the Vitis IDE comes with options for profiling, which can give us more detailed and precise results, based on which we can determine what to accelerate and if it is possible at all.

The non-intrusive TCF profiler does not require any additional BSP or application settings. The only required action is to configure and run the profiler while the application is running and the results will be viewed once available. In terms of configuration, sample aggregation and stack tracing are needed to view which functions had the most collected samples and where they were called from to determine what code segment should be accelerated.

The profiling report consists of the function name, percentage of samples collected in the function and its child calls (inclusive) and percentage of samples collected in the function without child calls (exclusive). The following table shows reduced report of only the most important functions that were enough for designing the hardware accelerator.

Function	Inclusive (%)	Exclusive (%)
_start	100	10.5
ecp_mul_comb	98.3	0
ecp_mul_comb_after_precomp	98.3	0
mbedtls_mpi_mul_mpi	92.1	3.12
ecdsa_sign_restartable	91.3	0
main	89.5	0
mbedtls_ecdsa_sign	89.5	0
ecp_mul_restartable_internal	86	0
ecp_mul_comb_core	84.2	0
mbedtls_ecp_mul_restartable	80.7	0
mbedtls_mpi_mul_mod	80.7	0
mbedtls_mpi_core_mla	50.0	50.0
ecp_add_mixed	48.2	0
ecp_modp	45.6	0
ecp_mod_p256k1	35.1	0

Table 5.1: Profiling results.

As can be seen in the table, the `mbedtls_mpi_core_mla` that implements generic integer multiplier used 50% of the samples exclusively, which indicates the need for a faster multiplier. The stack trace (not shown in the table) also indicates, that the multiplier is mostly called from function `mbedtls_mpi_mul_mod`, which implements modular exponentiation and for this reason it was selected as a reference function for hardware acceleration.

5.7 Hardware acceleration

Based on profiling results, we have been able to find the most demanding function within the ECDSA algorithm, which could potentially be accelerated and thus bring performance improvements. The implementation of multiplication within MbedTLS is based on decomposition into partial numbers, which are sequentially multiplied together in a loop until all of them are processed. Values in MbedTLS are stored in an array of integers, which have different size based on the used platform. On the Microblaze, which is a 32-bit architecture, the array consists of 32-bit unsigned integers. The multiplication algorithm is generic as it can process input arrays of variable size. Additionally, the multiplier is called mostly from function that processes modular exponentiation, which means that the modulo operation is executed next. Another important discovery is that the modulo function does not use division at all and it is implemented as a modular reduction instead. This fact explains why there was no additional performance improvement with integer divider feature enabled within the Microblaze. Instead of division, the modular reduction uses only multiplication and addition to get the result as fast as possible.

The process of creating a custom IP begins with the multiplier. Vivado HLS provides special classes for defining arbitrary precision data types known as `ap_int` for signed integers or `ap_uint` for unsigned integers. Operations with these data types are defined as well and a simple 256x256 multiplication can be written with a single line. However, a problem arises during synthesis, which reports insufficient DSP48 resources on the Zynq ZC702 FPGA. For this reason, it would not be possible to place the resulting board design with this multiplier on the board.

A different approach had to be taken to reduce the usage of DSP48 elements. The Karatsuba multiplication technique [14] has been implemented as a replacement, which decomposes both 256-bit input variables (A and B) into 128 most significant bits (Al and Bl) and 128 least significant bits (Ar and Br):

$$A = 2^{128}Al + Ar$$
$$B = 2^{128}Bl + Br$$

In terms of code implementation, the multiplication function has following declaration:

```
static void mul(uint32_t X[16], uint32_t A[8], uint32_t B[8]);
```

Listing 2: Multiplication function declaration

The input/output argument X and input arguments A and B of the function are arrays of 32-bit values, so the 128-bit numbers must be composed instead of decomposed. The `HLS unroll` pragma can be used to unroll the loop and load the values in parallel:

```
uint128_t AA[2];
uint128_t BB[2];

for(uint32_t i = 0; i < 4; i++)
{
    #pragma HLS unroll
    uint32_t lowIdx = 32 * i;
    uint32_t highIdx = (32 * (i + 1)) - 1;

    AA[1].range(highIdx, lowIdx) = A[i + 4];
    BB[1].range(highIdx, lowIdx) = B[i + 4];

    AA[0].range(highIdx, lowIdx) = A[i];
    BB[0].range(highIdx, lowIdx) = B[i];
}
```

Listing 3: Loading input values

At this point we have obtained a total of 4 values of 128-bit length and to calculate the 512-bit result denoted as X, a total of 4 multiplications, 3 additions and 3 bit shifts can be used:

$$X = 2^{256}AlBl + 2^{128}AlBr + 2^{128}ArBl + ArBr$$

Even though the above equation is viable, it is not very efficient. The Karatsuba algorithm can be further optimized by decreasing the number of multiplications and use additions (or subtractions) instead, which generally use less cycles. The addition of $AlBr$ and $ArBl$ can be rewritten into the following form:

$$AlBr + ArBl = (Al + Ar)(Bl + Br) - AlBl - ArBr$$

The final 512-bit result X can then be calculated in the following manner:

$$X = 2^{256}AlBl + 2^{128}((Al + Ar)(Bl + Br) - AlBl - ArBr) + ArBr$$

It is important to note that the addition can cause overflow by producing a carry bit, so additional data types must be defined to accommodate the carry bit if it is generated. The following code snippet demonstrates the Karatsuba multiplication algorithm on decomposed 256-bit values:

```
// Multiply higher and lower 128 bits (loop is used to decrease DSP48 usage)
uint256_t AB[2];
for (int i = 0; i < 2; i++)
{
    AB[i] = AA[i] * BB[i];
}

// Add higher and lower 128 bits, need 129 bit output value in case of carry
uint129_t sumAA = AA[1] + AA[0];
uint129_t sumBB = BB[1] + BB[0];

// Last multiplication of the sums according to the
// Karatsuba multiplication technique (258-bit result)
uint258_t mid = (sumAA * sumBB) - AB[1] - AB[0];
```

Listing 4: Karatsuba multiplication

At this stage, the output number `X` can be composed from array `AB` and value `mid`:

```
// Low 128 bits of ArBr
uint128_t low = AB[0].range(127, 0);
// High 128 bits of ArBr + low 128 bits of (Al + Ar)(Bl + Br) - AlBl - ArBr
uint129_t lowMid = AB[0].range(255, 128) + mid.range(127, 0);
// Low 128 bits of AlBl + high 128 bits of 'mid' + carry
uint256_t highMid = AB[1].range(127,0) + mid.range(257, 128) + lowMid[128];
// High 128 bits of AlBl + carry
uint128_t high = AB[1].range(255, 128) + highMid.range(255, 128);

// Final result
uint256_t XX[2];
XX[0].range(127, 0) = low;
XX[0].range(255, 128) = lowMid.range(127, 0);
XX[1].range(127, 0) = highMid.range(127, 0);
XX[1].range(255, 128) = high;

// Store results
for(uint32_t i = 0; i < 8; i++)
{
    #pragma HLS unroll
    uint32_t lowIdx = 32 * i;
    uint32_t highIdx = (32 * (i + 1)) - 1;

    X[i] = XX[0].range(highIdx, lowIdx);
    X[i + 8] = XX[1].range(highIdx, lowIdx);
}
```

Listing 5: Storing the result

The implementation of modular reduction in MbedTLS remains unchanged in the HLS except for usage of predefined arbitrary precision integers. Similarly to Karatsuba multiplication, the algorithm decomposes the 512-bit input value X into Xl and Xr. Both values are used in one multiplication and one addition, however, two passes of the following algorithm are done due to the possibility of carry:

$$X = 2^{256}Xl + Xr$$
$$M = Xl \times R$$
$$X = Xr + M$$

As the modular exponentiation algorithm is done, the only remaining thing is the communication interface that will be used to pass both input arrays and retrieve the output array. The AXI4-Lite interface has been chosen as an appropriate protocol for our needs as the data volume is relatively small. The synthesis process supports pragmas that can be used to select the requested interface for each function argument and the return value.

The hardware accelerator block for modular exponentiation has been implemented as multi-purpose. The last argument of the hardware acceleration function in HLS is the variable `op`, which can be used to select the required arithmetic operation(s). Based on its value, either multiplication (`OP_MUL`), modulo (`OP_MOD`) or both (`OP_MULMOD`) are calculated for input operands `A` and `B` and the input/output operand `X`.

```
void accelerator(uint32_t X[16], uint32_t A[8], uint32_t B[8], uint32_t op)
{
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=X bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=A bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=B bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=op bundle=CTRL

    if(op == OP_MUL || op == OP_MULMOD)
    {
        mul(&X[0], &A[0], &B[0]);
    }

    if(op == OP_MOD || op == OP_MULMOD)
    {
        mod(&X[0]);
    }
}
```

Listing 6: Hardware accelerator function

At last, the test bench has been created to verify the functionality of the algorithm. The outputs of original MbedTLS implementation and our functions were compared to detect any changes. The code has been synthesized and its FPGA resource usage with latency estimations reviewed before running the co-simulation (results of synthesis can be seen in section 6.5). Finally, the package containing the IP with its corresponding drivers has been exported to be used in Vivado Design Suite for further integration into the design.

Chapter 6

Evaluation

The most important output of this work are the final performance and memory usage results of the ECDSA signature algorithm implemented on a FPGA. First of all, many different configurations of the Microblaze soft-core processor had been timed to determine which features had an impact on the algorithms execution time. On software side, the Microblaze compiler flags were modified and the MbedTLS feature selection was changed to find an ideal compromise between performance and memory usage while maintaining security. The Merkle tree for storing collected hashes including the SHA256 hashing algorithm itself were also evaluated to decide if the compromise of using this feature is acceptable. Afterwards, the implemented code had been thoroughly profiled to identify the function with most processor cycles being used. Based on those profiling results, a dedicated hardware accelerator had been designed, synthesized and added to the overall board design to be used to further improve the computing potential. This chapter will focus on detailed profiling data that were gathered during implementation phase and were crucial for making further decisions to create the final solution.

6.1 Performance

At the early stage of implementation, it was important to find out the full computing potential of the Microblaze processor for ECDSA signature algorithm using a 256-bit key. Each Microblaze feature selected in Vivado Design Suite was timed independently and was either kept enabled if it had a positive impact or removed otherwise before testing a different configuration. Following tables show different configurations and their respective average execution times of the ECDSA signature algorithm with curve `SECP256K1` using MbedTLS library.

Clock input frequency (MHz)	Time (ms)
50	1472
100	741
125	592

Table 6.1: ECDSA on Microblaze with different input clock frequencies.

Additional features	Time (ms)
32-bit integer multiplier (MUL32)	269
64-bit integer multiplier (MUL64)	592
MUL32 and Basic FPU	269
MUL32 and Barrel Shifter (BSH)	91
MUL32, BSH and Integer Divider (IDIV)	91
MUL32, BSH and Machine Status Register (MSR)	91
MUL32, BSH and Branch Target Cache (BTC)	88
MUL32, BSH, BTC, Data and Instruction caches	88

Table 6.2: ECDSA on Microblaze optimized with additional features (125 MHz).

Based on table 6.1, the clock frequency is in linear relation to the execution time (doubling the input clock frequency from 50 to 100 MHz almost halved the execution time). This is expected behavior as with higher clock the processor is capable of processing instructions faster in the same period of time.

The effect of adding different features to Microblaze is shown in table 6.2. ECDSA algorithm is based on the mathematical concept of modular exponentiation and is expected to obtain performance boost when an integer multiplier is enabled. On the other hand, the lack of floating point operations resulted in no improvement when a floating point unit (FPU) was added. Another major impact on the performance was caused by adding the barrel shifter for bit shifting[1] and also a minor improvement by enabling branch target cache for branch prediction in conditions. Every other feature had no impact on the execution time of the ECDSA signature algorithm.

The MbedTLS library supports multiple elliptic curves available for usage in ECDSA. The only viable options are 256-bit keys, since they are secure enough to be used, but do not take as much memory space as larger key sizes. Performance evaluation of different elliptic curves is shown in the following table.

Elliptic curve	Execution time (ms)
`SECP256K1`	88
`SECP256R1`	435
`BP256R1`	528

Table 6.3: Comparison of different supported elliptic curves.

Additionally, the SHA256 algorithm has also been evaluated since it is used in the Merkle tree implementation. All gathered data are shown in the following table.

Compiler optimization level	Execution time (μs)
None (`-O0`)	130
Basic (`-O1`)	53
Moderate (`-O2`)	57
Best performance (`-O3`)	61
Lowest code size (`-Os`)	46

Table 6.4: SHA256 algorithm in different compiler optimization levels.

[1] https://www.d.umn.edu/~gshute/logic/barrel-shifter.html

The table above displays unexpected behavior as the `-Os` level provides 30% better performance over the `-O3` option, which should yield the best performance. For this reason, further evaluation of the Merkle tree had been done with the `-Os` level enabled.

Algorithm	Execution time (μs)
Single SHA256	46
Merkle tree with 8 stored SHA256 hashes	646
Merkle tree with 16 stored SHA256 hashes	1372

Table 6.5: The Merkle tree optimization algorithm execution time.

6.2 Memory consumption

After performance evaluation was done, the next important part was optimization of memory space consumption by the created software package. The MbedTLS library was minimized by keeping only the necessary features, but a big impact on memory space had been selection of the compiler optimization level, which can be seen in the following table.

Compiler optimization level	Memory space consumption (B)	Execution time (ms)
None (`-O0`)	54056	232
Basic (`-O1`)	38400	101
Moderate (`-O2`)	47656	98
Best performance (`-O3`)	54992	88
Lowest code size (`-Os`)	35400	102

Table 6.6: Software package for different compiler optimization levels.

As can be seen, the `-O3` optimization level brings the best performance, but takes the most memory space. This software package only includes the ECDSA algorithm without the implementation of ChaCha20 PRNG and the logic for gathering entropy. At this point, any of the available compiler levels can be used as they can all fit within the 64 KB memory block of the Microblaze. However, the addition of aforementioned features changes the situation as they noticeably increase the package size.

Compiler optimization level	Memory space consumption (B)	Execution time (ms)
None (`-O0`)	68464	232
Basic (`-O1`)	50328	101
Moderate (`-O2`)	59272	98
Best performance (`-O3`)	68328	88
Lowest code size (`-Os`)	46800	102

Table 6.7: Software package with added entropy collection and ChaCha20 PRNG.

The execution time remains unchanged as the entropy collection algorithm is not executed frequently and the ChaCha20 pseudorandom number generator is quite efficient. The package size increase for all optimization levels ranges from 25% to 33%. Both `-O0` and `-O3` optimization levels can no longer fit on a 64 KB memory block and the largest 128 KB configuration is required.

6.3 Hardware acceleration

After the integration of the hardware accelerator, the performance improvement is significant over the optimized Microblaze configuration, as it yields 2.5x decrease in execution time. The results of final solution with integrated accelerator can be seen in the following table.

Compiler optimization level	Memory space consumption (B)	Execution time (ms)
None (`-O0`)	70888	94
Basic (`-O1`)	71056	40
Moderate (`-O2`)	71704	35
Best performance (`-O3`)	80720	33
Lowest code size (`-Os`)	48688	40

Table 6.8: Software package with entropy, ChaCha20 and custom hardware accelerator.

Every configuration except for the `-Os` level can not fit on a 64 KB memory block on the Microblaze. The decision between the best possible performance and lowest memory usage must be made based on the specific situation. If every bit of performance is necessary and memory requirements for additional code stay below 128 KB, the `-O3` level is the best option. However, if the compromise of 20% lower performance is acceptable and the final design requires less than 64 KB of memory, the `-Os` compiler level is the only viable option.

The last table in this section overviews the differences between the original Microblaze configuration, optimized Microblaze configuration and the hardware accelerated version.

Compiler optimization level	Original	Optimized	Hardware accelerated
None	759	232	94
Basic	629	101	40
Moderate	620	98	35
Best performance	592	88	33
Lowest code size	636	102	40

Table 6.9: Overview of the differences between configurations of Microblaze.

6.4 Verification

Even though the verification process is expected to be executed on a remote server, the algorithm had been evaluated as well for reference in case it was considered for usage. Following tables compare the execution times and package sizes for different optimization levels and show the impact of the custom hardware accelerator.

Compiler optimization level	Memory space consumption (B)	Execution time (ms)
None (`-O0`)	59528	786
Basic (`-O1`)	43056	349
Moderate (`-O2`)	52368	325
Best performance (`-O3`)	60288	309
Lowest code size (`-Os`)	40008	349

Table 6.10: Execution time and package sizes of the ECDSA verification algorithm.

Compiler optimization level	Memory space consumption (B)	Execution time (ms)
None (`-O0`)	76880	286
Basic (`-O1`)	75720	137
Moderate (`-O2`)	76416	116
Best performance (`-O3`)	86024	113
Lowest code size (`-Os`)	53304	134

Table 6.11: Execution time and package sizes of the ECDSA verification algorithm with hardware acceleration.

Table 6.10 indicates that the verification algorithm requires a lot more computing power in order to be usable in real time. If the process was ever executed on the Microblaze for verification of signed frames from a camera, it could only verify 3 frames per second at best, which is considered unusable for current standards. Additionally, the memory space consumption is increased over the original sizes shown in table 6.6, which is mainly due to the need of a 50% larger stack and 2x larger heap in order for the algorithm to successfully execute.

However, if the verification algorithm was necessary on a FPGA, it should be used with the hardware accelerator enabled. The performance gain of 2.5x makes it more usable as it is now able to verify almost 9 signatures per second. In terms of memory usage, the results are again very similar to the package in table 6.8, which means that only the `-Os` can fit into a 64 KB block.

6.5 Synthesis

During implementation of the hardware accelerator, the resource usage of the FPGA had to be monitored in order to successfully integrate it into the overall design. Some implementations were very fast, but had used more DSP48 resources than available, which led to implementation of explicit delays and other code changes. This affected the overall latency, but decreased the DSP48 usage and the block placement became possible. The final solution latency and utilization reports can be seen in following tables.

Latency (cycles)		Latency (ns)		Interval (cycles)	
min	max	min	max	min	max
2	78	16	624	2	78

Table 6.12: Latency summary.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	97
Instance	6	31	7705	3847
Multiplexer	-	-	-	368
Register	-	-	1353	-
Total	6	31	9058	4312
Available	280	220	106400	53200
Utilization (%)	2	14	8	8

Table 6.13: Utilization estimates.

Chapter 7

Conclusion

This thesis is a part of a larger project `SECUSEN`, which focuses on data security by using asymmetric signature algorithms and timestamps. Its main goal is to implement a secure and efficient solution for signing raw image data from a camera implemented on a FPGA. The best way to accomplish this task is to cryptographically sign the data as close to their source as possible in order to ensure their integrity, authenticity and non-repudiation.

The main goal of this work is to find an optimal solution to this problem, which includes the selection of a secure asymmetric algorithm, key type and size, implementation of randomness on a FPGA and the research of processing units capable of executing such algorithms in real time.

The ECDSA signature algorithm has been chosen due to the usage of small keys. Its implementation within MbedTLS focuses on embedded systems, including the Microblaze soft-core processor that can be synthesized on a FPGA. The curve `SECP256K1` supported in MbedTLS provides the best performance among all available curves and has been selected for the final solution (see table 6.3). In order to keep the private key secret, a pseudorandom number generator must be present to generate a random nonce for each signature. The ChaCha20 PRNG has been chosen for its cryptographic security, which is periodically seeded with data from an entropy source collected by the ADC and the temperature sensor on Zynq ZC702.

In addition to the support in MbedTLS, the Microblaze processor is configurable based on specific requirements of the algorithm, which is why it is used as the main processing unit. A combination of specific features and added optimizations resulted in a solution, that is usable in real time (see table 6.6). Thanks to available debugging and profiling tools, additional performance improvements were made to the algorithm in a form of hardware acceleration. The formula for modular exponentiation has been implemented in the Vivado HLS, synthesized and integrated into the MbedTLS library.

As shown in table 6.8, the acceleration unit had a performance impact of 2.5x over the optimized Microblaze implementation, which lowered the final execution time from 88 to 33 milliseconds per signature. In terms of future work, more potential improvements could be made with the Edwards family of elliptic curves and the EdDSA algorithm.

Bibliography

[1] Texas Instruments. *Principles of Data Acquisition and Conversion* [online]. 1994. April 2015. Available at: https://www.ti.com/lit/an/sbaa051a/sbaa051a.pdf.

[2] Kuon, I. and Rose, J. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Feb 2007, vol. 26, no. 2, p. 203–215. DOI: 10.1109/TCAD.2006.884574. ISSN 1937-4151.

[3] Savarese, C. and Hart, B. *The Caesar Cipher* [online]. [cit. 2023-05-09]. Available at: http://www.cs.trincoll.edu/~crypto/historical/caesar.html.

[4] Deepthi, K. and Singh, K. Cryptanalysis of Salsa and ChaCha: Revisited. In: *Mobile Networks and Management*. Springer International Publishing, May 2018, p. 324–338. DOI: 10.1007/978-3-319-90775-8_26. ISBN 978-3-319-90774-1.

[5] Mohammed Nazeh Abdul Wahid, B. E. and Marwan, M. A Comparison of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish for Guessing Attacks Prevention. *Journal of Computer Science Applications and Information Technology* [online]. august 2018, vol. 4, [cit. 2023-01-15]. DOI: 10.15226/2474-9257. ISSN 2474-9257. Available at: https://symbiosisonlinepublishing.com/computer-science-technology/computerscience-information-technology32.pdf.

[6] Johnson, L. Chapter 11 – Security Component Fundamentals for Assessment. In: *Security Controls Evaluation, Testing, and Assessment Handbook*. Boston: Syngress, 2016, p. 531–627. DOI: https://doi.org/10.1016/B978-0-12-802324-2.00011-7. ISBN 978-0-12-802324-2.

[7] Menezes, A. J., Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*. 1st ed. USA: CRC Press, Inc., 1996. ISBN 978-0-8493-8523-0.

[8] Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C. et al. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*. may 2015, vol. 77, p. 493–514. DOI: 10.1007/s10623-015-0087-1.

[9] Diffie, W. and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory*. November 1976, vol. 22, no. 6, p. 644–654. DOI: 10.1109/TIT.1976.1055638. ISSN 1557-9654.

[10] Barr, M. and Massa, A. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, Inc., 2006. ISBN 978-0-596-00983-0.

[11] Kuon, I., Tessier, R. and Rose, J. FPGA Architecture: Survey and Challenges. *Foundations and Trends® in Electronic Design Automation*. 2008, vol. 2, no. 2, p. 135–253. DOI: 10.1561/1000000005. ISSN 1551-3939.

[12] Zhu, J. and Dutt, N. CHAPTER 5 – Electronic system-level design and high-level synthesis. In: Wang, L.-T., Chang, Y.-W. and Cheng, K.-T. T., ed. *Electronic Design Automation*. Boston: Morgan Kaufmann, 2009, p. 235–297. DOI: 10.1016/B978-0-12-374364-0.50012-6. ISBN 978-0-12-374364-0.

[13] Mohamed Khalil Hani, M. N. M. Hardware Acceleration of OpenSSL cryptographic functions for high-performance Internet Security. In: *2010 International Conference on Intelligent Systems, Modelling and Simulation*. 2010 [cit. 2023-01-19]. DOI: 10.1109/ISMS.2010.89.

[14] Weimerskirch, A. and Paar, C. Generalizations of the Karatsuba Algorithm for Efficient Implementations. *IACR Cryptology ePrint Archive*. january 2006, vol. 2006, p. 224.

[15] Jiang, J.-H. R. and Devadas, S. CHAPTER 6 – Logic synthesis in a nutshell. In: Wang, L.-T., Chang, Y.-W. and Cheng, K.-T. T., ed. *Electronic Design Automation*. Boston: Morgan Kaufmann, 2009, p. 299–404. DOI: 10.1016/B978-0-12-374364-0.50013-8. ISBN 978-0-12-374364-0.

[16] Merkle, R. C. A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C., ed. *Advances in Cryptology — CRYPTO '87*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, p. 369–378. DOI: 10.1007/3-540-48184-2_32. ISBN 978-3-540-48184-3.

[17] Barker, E. *Recommendation for Key Management* [online]. 2020 [cit. 2023-01-18]. DOI: 10.6028/NIST.SP.800-57pt1r5. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf.

[18] Aranha, D., Novaes, F., Takahashi, A., Tibouchi, M. and Yarom, Y. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. October 2020, p. 225–242. DOI: 10.1145/3372297.3417268.

[19] O'Neill, M. E. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, september 2014.

[20] Bouillaguet, C., Martinez, F. and Sauvage, J. Practical seed-recovery for the PCG Pseudo-Random Number Generator. *IACR Transactions on Symmetric Cryptology*. september 2020, p. 175–196. DOI: 10.46586/tosc.v2020.i3.175-196.

[21] Rivest, R. L. and Kaliski, B. RSA Problem. In: Tilborg, H. C. A. van, ed. *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2005, p. 532–536. DOI: 10.1007/0-387-23483-7_363. ISBN 978-0-387-23483-0.

[22] Lines, A. Asynchronous interconnect for synchronous SoC design. *IEEE Micro*. Jan 2004, vol. 24, no. 1, p. 32–41. DOI: 10.1109/MM.2004.1268991. ISSN 1937-4143.

Appendix A

Contents of the attached SD card

- **hls/** – Implemented hardware accelerator in HLS
- **vitis/** – Microblaze software package for accelerated ECDSA
- **vivado/** – Block design in Vivado Design Studio