Exploring Feasibility of FPGAs in Implantable Medical Devices

MSc Computer Engineering

David Veselka

Delft University of Technology

TUDelft

Exploring Feasibility of FPGAs in Implantable Medical Devices

by

David Veselka

to obtain the degree of Master of Science
at Delft University of Technology,
to be defended publicly on Wednesday July 28, 2021 at 11:00 AM.

Student number:	4280199	
Project duration:	March 23, 2020 – July 28, 2021	
Thesis committee:	Prof. dr. ir. J. S. S. M. Wong	TU Delft, supervisor
	Prof. dr. ir. T. G. R. M. van Leuken	TU Delft
	Dr. ir. R. Bishnoi	TU Delft
	Prof. dr. ir. C. Strydis	Erasmus MC
	Ir. M. A. Siddiqi	Erasmus MC, daily supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Implantable Medical Devices (IMDs) are deployed in patients to treat a range of medical conditions. Technological advancements have enabled manufacturers to fit IMDs with specialized hardware that accelerates compute-intensive medical therapies next to a software-run host processor. However, mostly hardware acceleration is found in the form of ASIC peripherals next to a host processor in state-of-the-art IMDs, while low-power FPGAs could provide a comparable performance gain with the added benefit of the upgradability of functionality. Existing literature about low-power FPGAs focuses on new algorithms or performance improvements, while largely ignoring power and energy analysis, the latter being the most limiting factor in the IMD environment. This thesis investigates under what conditions FPGAs could be added to IMDs by developing two use cases: an FPGA securing wireless communication, and accelerating a neural network aiding medical therapies that depend on pattern detection. These cases are evaluated on FPGA, eFPGA and MCU with ASIC peripheral platforms, from which performance, energy usage and prospected IMD battery life is derived. On one end, it was found that AES encryption used 4.4 times the energy of an MCU hardware-accelerated implementation while being 17% slower. However, employing lightweight ciphers on the FPGA closes this gap. Furthermore, adding an FPGA results in only a 7.5% decrease in battery life when the FPGA is shut off during idling to combat its high static current draw. Running an FPGA-accelerated neural network is feasible if the active time is 6.5 minutes per day. With weekly recharging, continuous monitoring is possible. Using an eFPGA, which is an embedded FPGA fabric integrated within an MCU, results in using only 12% to 21% of the FPGA package area and is almost 2 times as energy efficient under 2 minutes daily usage as an FPGA. As FPGAs in IMDs is a novel field, research was done in legal regulation of IMDs, where it was found that existing regulations on software devices also applies to FPGAs. Therefore, all obstacles of the technical and legal kind have been removed that hold IMD manufacturers from using FPGAs in their devices.

David Veselka
Delft, July 2021

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AES	Advanced Encryption Standard
AHB	AMBA High-performance Bus
ANN	Artificial Neural Network
ASIC	Application-specific integrated circuit
CNN	Convolutional neural network
FDA	Food and Drug Administration
FPGA	Field-programmable gate array
FSM	Finite-state machine
HDL	Hardware description language
HW	Hardware
IMD	Implantable medical device
IP	Intellectual property
LUT	Lookup Table
MCU	Microcontroller unit
PCB	Printed circuit board
RTL	Register-transfer level
SW	Software
UART	Universal asynchronous receiver-transmitter

List of Figures

List of Tables

Acknowledgements

First of all, I am grateful to God for the grace and perseverance that were necessary to complete this thesis.

From Erasmus MC, I would like to thank prof. dr. Christos Strydis for overseeing the technical part of my thesis and tracking and steering my progress where needed. Likewise, I thank Ali Siddiqi for his daily supervision and being always available to answer my questions about any part of my research. With their help, the quality and efficiency of the conducted research has greatly improved.

I would also like to thank prof. dr. Stephan Wong and dr. Rajendra Bishnoi from TU Delft for their supervision and helping me to keep up the pace in thesis writing, specifically towards the end. Without their help, I would definitely not having finished my thesis by now.

I would also like to thank the employees from Menta S.A.S. for their generosity in giving me time to use their eFPGA design tools, enabling small eFPGA energy analysis which is novel in literature. Especially thanks to Catherine le Lan, who gave me technical support in weekly meetings and more when it was needed.

Finally, I must express my very profound gratitude to my friends, family and my girlfriend for providing me with unfailing support and encouragement throughout the process of researching and writing this thesis, without whom this would not have been possible.

1 Introduction

1.1. Implantable Medical Devices

Implantable Medical Devices (IMDs) are deployed in patients to treat a range of medical conditions. Like other electronic devices, IMDs benefit from technological advancements that make extending features and functionality possible. Likewise, many new methods for examining and treating health conditions have emerged, with implants serving as pacemakers, neurostimulators (Figure 1.1) and drug administering devices. Since the 1990s, IMDs have featured MCUs at their core [1]. This enables recent devices to have features such as software upgradability [2], wireless connectivity for live monitoring and the possibility for more computational intensive algorithms incorporated in neurostimulators.

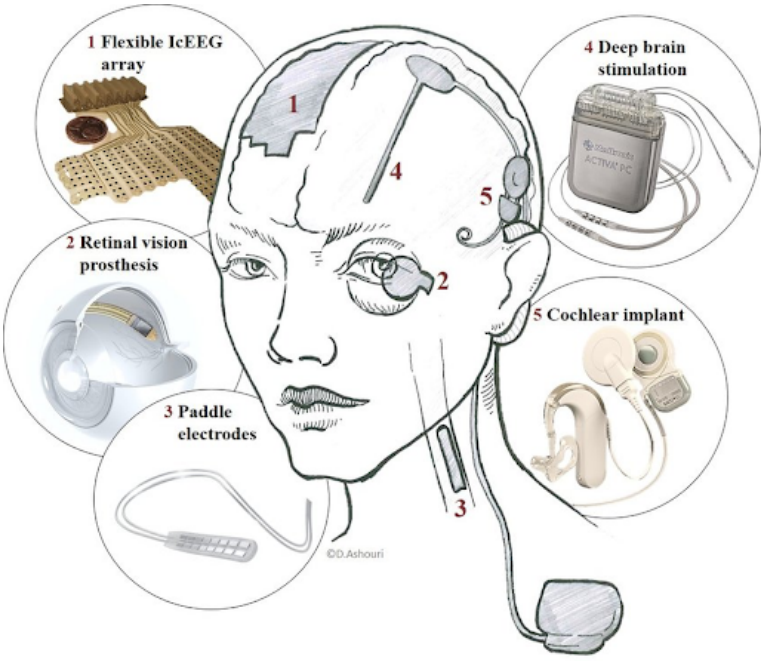


Figure 1.1: Various neural implants and the location where they interface the central nervous system. [3]

First, the incorporation of a wireless channel brings a number of advantages. Through a communication channel, interventions like upgrading firmware, extending functionality and correcting bugs can be performed without physical hardware replacement. Remote monitoring via a specific reader device or even via a smartphone is already implemented in recent IMDs [2]. A feature added in recent IMDs is wireless power transfer which allows the IMD to harvest energy, saving on battery charge and extending the lifetime of the implant [4]. These advantages can be summarized in a general result: not

having to perform surgery if one of the named interventions needs to be performed.
However, having wireless connectivity also has negative consequences. When active, wireless communication is responsible for the major part of the total power consumption of an IMD. To avoid rapid battery depletion, communication duration has to be kept to a minimum and energy efficient protocols have to be used. Additionally, a wireless link opens the door to potential adversaries that can try to compromise the implant. Due to hardware restrictions, even recently developed implants did not impose security on their wireless links. This poses a risk to getting hijacked with inexpensive equipment [5]. Even if some form of security is implemented, weaknesses in a standard protocol like Bluetooth Low Energy (BLE) cause many devices to be vulnerable [6]. Therefore, research is needed on the feasibility of making an IMD both energy efficient and secure.

Second, with the rapid increase in computational performance-per-watt in the semiconductor industry, advanced algorithms become feasible to run on low-power hardware. Problems like seizure detection and prediction can be integrated in IMDs when using specialized hardware [7]. Moreover, artificial neural networks (ANNs), renowned for their pattern detection ability superior to conventional algorithms, are within reach [8]. Integrating ANNs within IMDs would open a realm of possibilities for medical treatments that were computationally infeasible before. But, ANNs are generally known for their high arithmetic intensity, thus profiting from hardware acceleration for gains in performance and energy efficiency. As the IMD environment is tightly constrained, especially in terms of available energy, hardware acceleration would most likely improve feasibility of ANNs in IMDs.
However, no published literature combines hardware-accelerated artificial neural networks with the tightly constrained environment of IMDs, featuring seizure prediction or equivalent medical therapies. Neural networks are shown to operate on a low-power FPGA [9], but no analysis is made with regard to prospected IMD battery life and the impact of FPGA hardware acceleration thereupon. Therefore, additional research is needed on the conditions under which an ANN can be deployed on an FPGA-equipped IMD.

In the following sections of this chapter, the main trends and challenges in IMD hardware will be highlighted and focus will be put on the use of FPGAs alongside standard microprocessors found in modern IMDs. Due to the reconfigurable nature of FPGAs, algorithms can be parallelized more than CPUs at a similar footprint and energy consumption. FPGAs will be considered for two purposes: securing a wireless communication interface and as a functional unit that accelerates medical therapy, with primary focus on the communication interface. The goal is not to reinvent the wheel in the sense that the processor, algorithms and peripheral interfaces will all be built from the ground up. Instead the state-of-the-art in wireless IMD security and functional algorithms will be reviewed and a selection implemented on reconfigurable hardware to gain performance and power consumption figures. Next to technical evaluation, research on the legal certification process is performed to determine whether an FPGA-equipped IMD needs extra legislative steps to be allowed on the market.

1.2. Problem statement

An IMD with wireless connectivity should have strong enough security mechanisms to avoid security breaches. Subsequently, functionality and security hardware has to adhere to the tightly energy constrained environment. Current IMDs, however, are mostly limited by underpowered hardware and battery life restrictions. ASICs and even FPGAs have been proposed as auxiliary devices in IMDs to provide more computing power and higher energy efficiency than CPUs. ASICs have a high upfront cost and do not have the possibility to have their functionality fundamentally changed without physical access. FPGAs have the potential to overcome these shortcomings, but no research provides a complete picture on FPGAs in IMDs in terms of functionality while taking into account medical certification, detailed energy consumption and the possibility of hardware reconfiguration without physical access. The problem statement that is proposed is therefore formulated as follows:

Can FPGAs be used as a reconfigurable fabric within IMDs?

The novelty of this research direction lies in the completeness of exploring the feasibility of IMD FPGAs, not only on functionality, energy consumption or reconfigurability but combining all three, among

other factors. By having a complete picture, IMD manufacturing companies could consider FPGAs as most beneficial for a certain use case. This research will try to uncover in what cases an FPGA provides clear advantages for usage in IMDs.

1.2.1. Research questions

To be able to answer the problem statement properly, it has to be broken down in several research questions. These questions are tied to different aspects that determine feasibility of FPGA-equipped IMDs. Classical properties like performance gain and reconfigurability drive the interest in FPGAs, but putting them in IMDs is only a good idea if energy demand is not too high and the corresponding performance gain is significant. Furthermore, potential legal regulation obstacles need to be removed to answer the problem statement positively.

The research questions are formulated as follows:

1. **Can FPGA fabric be used in IMDs in terms of energy and execution time?**
   The main features for wanting to integrate FPGAs in IMDs is the prospected performance gain with parallelizable algorithms together with the flexibility of reconfigurability. For this reason, we will evaluate performance differences compared to ASIC and CPU implementations common in modern IMDs. In addition, we will look at FPGA resource usage of all created hardware implementations to evaluate whether they fall within the resource count bound of our low-power reference FPGA [10]. Moreover, mainly constraining performance is the limited energy available from the small integrated battery, that is supposed to provide a multi-year operation time before needing to be replaced. These points will be addressed in the first research question.
2. **How big are the improvements of an eFPGA over an FPGA with regard to energy consumption and area?**
   Regarding area, the available physical size in an IMD is another determining requirement for eligible FPGAs. IMDs often have less than 10cm^2 of PCB area, ruling out adding FPGAs with a large package size. However, FPGA solutions exist that do not involve adding an extra physical chip. Instead, an embedded FPGA, eFPGA for short, is integrated onto the same silicon die as the host processor during manufacturing. This technology could possibly provide advantages in area and energy consumption which has a direct impact on IMD battery life. Furthermore, performance with regard to latency and throughput is greatly improved. For the area comparison, a look is taken at the respective silicon areas without surrounding components. In answering the second research question, the gains and benefits of eFPGAs compared to regular FPGAs will be examined to see whether eFPGAs contribute significantly to feasibility.
3. **In what cases is daily FPGA reconfiguration beneficial in IMDs?**
   A unique feature of FPGAs not found in regular microprocessors or ASICs is hardware reconfigurability. This provides the possibility to run any design with the performance of hardware implementations and with the ease of upgrading similar to software implementations. Designs can be swapped on the fly for extended functionality or obsolete implementations can be upgraded without having to resort to slower software implementations. However, where reconfiguration is trivial for high-performance platforms, it is not for the ultra-low-power domain of IMDs due to energy constraints. This leads to the third question.
4. **Having an FPGA-equipped IMD, what are the difficulties in aspect to legal certification?**
   Next, the certification process that all IMDs have to go through before being put on the market is analyzed. A federal regulatory body has to certify a complete device with its hardware and software. If FPGAs would be used in commercial IMDs, they would likely require to being certified for every hardware configuration that is used. If individual bitstreams can be separately certified, this would mean that the complete functionality of an IMD could be changed without a change in physical hardware. In answering the fourth question, the legal aspect is tackled.

1.2.2. Project Goal and Scope

The goal of of this thesis is to see in what areas (e)FPGAs, added to a traditional MCU-based IMD, are advantageous over MCU-only IMDs. Quantitative results will be given in terms of performance gain compared to traditional MCUs, energy consumption and expected battery life with and without FPGA fabric. Designing an algorithm or IMD hardware/software architecture from scratch is considered out of

scope for this thesis. Instead, most of the platforms and algorithms that are used in the conducted experiments are pre-existing and openly available. The result from this thesis will mainly be an overview on the most preeminent advantages that FPGAs will bring to the table in the resource-restricted environment of IMDs. Furthermore, the quantitative conditions under which FPGAs are feasible and beneficial in IMDs will be obtained by the conducted experiments.

1.3. Methodology

As all research questions are geared towards an IMD-specific environment, typical use cases for an IMD FPGA need to be defined. Noted in Section 1.1, two primary use cases will be emphasized in this thesis: securing wireless communication and accelerating medical therapy. By designing experiments that are based on these use cases, a realistic view of FPGA feasibility can be obtained. In the next subsections the two use cases will be explained in more detail.

1.3.1. Case 1: security primitives

As has been mentioned in this chapter, wireless connections pose a security threat if no security mechanism is in place. Recent IMDs that have security often hide the details about the scheme used in a 'security through obscurity' philosophy. The danger of this is that no external party can verify if the security scheme is robust. Several cryptographic primitives, specifically focused on the hardware and energy constraints of IMDs, have been proposed to be used in a freely available scheme [11] [12]. Also, complete cryptographic schemes have been developed to have an alternative for current obfuscated schemes [13] [14]. Hardware implementations of these open schemes and primitives have shown that custom hardware, as opposed to a general-purpose processor, provides a major advantage in execution time and energy consumption[15][16][17].

Running security primitives on FPGAs is not new. However, doing this in IMDs is not done before commercially, most likely due to added hardware complexity and a supposed increase in power consumption. In this case, several open security primitives are selected and implemented on IMD-fitting FPGA fabric, with a communication interface to connect to the host CPU. The well-known Advanced Encryption Standard (AES) [18] is taken as reference, together with a lightweight block cipher (SIMON) [19] and hashing algorithm (PHOTON) [20] for broader comparison. In addition, hardware communication interfaces are attached to all three blocks for an evaluation of a complete hardware block, as it would appear in a real IMD. Evaluating this case will give insight in under what conditions it would be beneficial to incorporate an FPGA into an IMD for secure communication.

1.3.2. Case 2: Artificial Neural Network

Inspired by biological neural networks, artificial neural networks (ANNs) are computing components that analyze and process information. With ANNs, complex problems can be solved that would be difficult by human standards. In the field of IMDs these networks could be deployed for advanced pattern detection in brain signals, for example to detect and predict incoming seizures [21, 22]. With ever-increasing computational power of modern hardware, artificial neural networks become accessible to smaller devices with a more stringent energy budget [23]. In literature and in this thesis, small neural networks that occupy <10k LUTs and are targeted to run on platforms consuming <10mW will be called tiny neural networks, TNNs for short. However, designing a TNN is a tedious task, let alone making one specifically for the aforementioned purpose, seizure prediction. As has been said in Section 1.2.2, this thesis is not design-centered. Instead, a readily-available implementation of a TNN will be evaluated on FPGA fabric in terms of energy and resource usage. However, most TNNs are only available as software implementation and those that exist in both hardware and software versions are mostly geared towards higher-end hardware. For a full analysis, see Section 2.3. As a result, a sample project from Lattice Semiconductor, Inc. will be loaded on a low-power FPGA from the same vendor. No equivalent software implementation is available for comparison unfortunately, but the hardware implementation can give a feasibility picture on its own. The main reason for not having a software-hardware comparison of a TNN, is because of unavailability: during the complete duration of this project, no TNN with both implementations was available to us. An explanation of this unavailability can be attributed to the novelty of TNNs in IMDs. To the best of our knowledge, no publicly available literature exists that presents a IMD-focused TNN. By incorporating this use case, a look is taken at feasibility of future

algorithms to be implemented in IMDs.

1.3.3. Evaluation

Both use cases will be elaborated on in a series of experiments. They consist, per use case, of comparing functionally identical FPGA and software implementations (with and without hardware acceleration) in terms of static and dynamic power draw, performance and most importantly, energy consumption. To finally determine whether an implementation is feasible for IMDs, typical usage scenarios are drafted. Through modeling idle and active usage, expected battery life is obtained for each experiment and platform. **Feasibility is hereby defined as: enabling an IMD battery life of 2.5 years or more, when no recharging option is available.**

1.4. Thesis structure

In Chapter 2, literature is explored and reviewed to gain background knowledge about recent technological trends in IMDs and motivation for the conducted experiments. Next, Chapter 3 outlines the selected experiment platforms and algorithms. Furthermore, designed hardware descriptions and experiment preparations are discussed. Chapter 4 contains results obtained from the experiments and evaluation on these results is done. Finally, Chapter 5 concludes this thesis and states the directions in which future work can be done.

2

Background

In Chapter 1, we stated a problem statement that is related to the feasibility of FPGAs in IMDs and motivated why we would like to focus on it. In order to have better understanding of this problem statement it is necessary to have a certain level of background knowledge about the following topics:

- **Ultra-low-power FPGA and eFPGA technology** is evaluated in Section 2.1 to explore the options in platform choice for hypothetical FPGA-equipped IMDs. Not only off-the-shelf FPGAs are considered, but embedded FPGAs (eFPGAs) are introduced that have promising power and area characteristics specifically for IMDs.
- **Cryptography for ultra-low-power hardware** enables secure communication of IMDs with the outside world. To run cryptographic primitives on low-power hardware, lightweight algorithms have been proposed and implemented in literature. In addition, existing standard primitives have been adapted for low-power suitability. Both categories and a general introduction to the different cryptographic primitive families will be discussed in Section 2.2, with focus on FPGA implementations.
- **Artificial neural networks and their application in low-power environments** has been a field of recent interest and is explored in Section 2.3. First, a general introduction to neural networks will be given to make the reader familiar with the matter. Thereafter, literature is reviewed focusing on hardware-accelerated low-power neural networks, with the goal of finding an FPGA implementation of an IMD-suitable network for our experiments.
- **Wireless energy harvesting** is determining for the ability to run algorithms within an IMD without using its integrated battery. As energy is a scarce resource in an IMD, measures have to be taken to prevent adversaries from purposely draining the battery. In Section 2.4, the state-of-the-art in energy harvesting technology for IMD-sized devices is explored to derive an upper bound in power consumption for IMD FPGAs.

2.1. Ultra-low-power FPGA and eFPGA technology

2.1.1. Processing hardware developments

Processing in traditional IMDs mainly employ two solutions: MCUs with a software driven processor and, more recently, MCUs with added hardware acceleration ASIC peripherals. MCUs allow for inexpensive manufacturing, as these devices tend to be mass produced. Furthermore, changing functionality or fixing bugs in its application software can be performed with minimal effort. However, with a very strict and small power envelope allowed in IMDs, performance tends to be rather lacking and functionality is limited to simple algorithms like pace making or uncomplicated neurostimulation [2]. On the other end, ASIC processing blocks have been proposed for IMDs to alleviate the main processor of compute-intensive tasks [24]. This way, execution time and energy costs are diminished. Hardware acceleration blocks within MCUs, designed as ASICs, are more power efficient, smaller and faster than equivalent software-only MCUs. If an IMD with an application specific device gets certified and is allowed to the market, it can be reused again in newer models which may shorten time-to-market as the possibility exists that no re-certification needs to happen. Among the disadvantages are: inflexibility in

Table 2.1: Ultra-low-power FPGAs used in literature

Ultra-low-power FPGAs used in literature							
FPGA model	**Reference**	**Type of usage**	**Clock speed (MHz)**	**Power (mW)**			**Energy usage (nJ)**
				Total	**Static**	**Dynamic**	
CycloneV 5CEBA4	[26]	Neurostimulation	7.8	43.86	40.14	3.72	-
IGLOO2 M2GL025				20.06	16.22	3.84	-
iCE40 HX-8K				3.62	2.24	1.38	-
XCS31400A	[15]	SHA-1 encryption	62.678	6.6	-	-	132000
iCE40	[27]	Sensor interface	-	0.12	-	-	-
IGLOO AGL10	[28]	Wireless message management unit	-	0.101	0.008	0.093	-
Custom	[29]	4-bit adder	-	0.0076	-	-	0.0029
iCE40			-	0.11	-	-	0.0038
IGLOO			-	0.5	-		0.0034
IGLOO AGLN250	[30]	Neuroprocessor for 32 channels @25ksps	6.4	5.19	-	-	-
IGLOO AGL250V2	[31]	Seizure detection unit	0.0052	0.11	-	0.11	-
IGLOO AGLN250	[8]	ANNs for person identification	24.26	7.579	0.079	7.5	27.2
IGLOO M1AGL600			22.13	15.281	0.131	15.15	20.9
IGLOO M1AGL1000			21.2	30.213	0.213	30	23.5
iCE40 UltraPlus UP5K			33.8	9.577	0.277	9.3	8.1

hardware and inability to do fundamental bug fixes or feature upgrades. Next, the initial cost in engineering effort is much higher than when using an FPGA for the same purpose. These reasons lead to the decision to mainly consider a third solution in this section: FPGA devices, which are emerging in low-power variants in recent years. FPGAs would be the ideal middle ground, retaining flexibility and offer a considerable performance gain which is direly needed. The possibility of completely changing its functionality by reconfiguration is especially useful in the IMD environment, where no physical access for multiple years is preferred, as surgeries have to be avoided whenever possible. It has to be noted however, that adding an FPGA to an IMD will possibly reduce battery life to the point where it is infeasible to have an FPGA-equipped IMD run for multiple years without recharging. Especially high static power is a well-known property of FPGAs. To investigate and show that FPGAs can be feasible in IMDs under certain conditions, typical FPGA use cases for IMDs are designed and used for power, area and performance measurements. As an addition, eFPGAs, short for embedded FPGAs, are considered, which can alleviate energy and area concerns that arise with adding an off-the-shelf FPGA to an IMD. Two main use cases are distinguished, which are presented in Sections 1.3.1 and 1.3.2 respectively. To the best of our knowledge, no paper considers an extensive energy analysis using IMD-suitable FPGAs. Instead, focus is often put on new algorithms or performance gains, mostly on ASICs or MCUs [25, 15]. By going through each of our cases, process hardware developments are put into an energy feasibility perspective.

2.1.2. FPGAs

To get a more concrete insight about the FPGAs that may be suitable to evaluate for IMDs, literature was explored to gain insight in the different FPGAs used for low-power purposes. Table 2.1 shows a selection of papers with their used FPGAs, their purpose and their energy and power consumption figures. The power and energy figures presented in Table 2.1 vary greatly with clock speed and application, so it is for indicative purposes only. From Table 2.1, it can be noted that two FPGA families are of particular interest: The iCE40 from Lattice and IGLOO from Microsemi. Other FPGAs have a static power consumption of more than 10mW, which will deplete an IMD battery very fast. As the IGLOO FPGA family was not accessible to us due to availability issues, the Lattice iCE40 is taken as the baseline FPGA in our experiments.

2.1.3. eFPGAs

eFPGAs, short for embedded FPGAs, have been showing up over recent years. They are marketed as being faster and more efficient than a traditional setup of an external physical FPGA connected to the MCU, because of integration of FPGA fabric inside the MCU. This could mean that no softcore has to be implemented in FPGA logic if needed, because a hard CPU can be available next to an eFPGA. For IMDs, this would result in a more efficient design that may offer a more powerful platform with a

Table 2.2: Low-power FPGA and eFPGA options

FPGA vs eFPGA options				
Manufacturer	**Evaluation board**	**LUTs/LEs**	**MACs**	**Technology node**
Achronix	Yes	up to 2.6M	not mentioned	TSMC 7nm FF, 12nm FFC, 16nm FFC
QuickLogic	Yes	1019	not mentioned	TSMC 65nm, 40nm, GF 65nm, 40nm, 22FDX
Menta	Yes	100-200k	up to 1000+	Any
Adicsys	No	100-100k	not mentioned	Any
FlexLogix	Yes	182000	560	Sandia 180nm, TSMC 40nm, TSMC 28/22nm, TSMC 16/12nm, GF 12nm
Microsemi IGLOO	Yes	100-3000	not mentioned	130nm
Lattice iCE40	Yes	640-5280	up to 8	40nm

small energy budget. eFPGAs are available as IPs and therefore simulations are normally done in an ASIC design flow. However, seeing that this process is very time intensive and out of scope for this thesis, our eFPGA experiments will be conducted using software design tools made available by eFPGA vendors. Physical measurements would be possible with evaluation boards, of which some vendors have units listed. Table 2.2 shows the major eFPGA manufacturers together with our chosen 'traditional' low-power FPGAs. Resource numbers indicated are for the available evaluation boards, as any resource number could be specified while designing an eFPGA. The QuickLogic and Menta boards are focused on small designs and would therefore be the best candidates to consider for IMDs, having a similar number of LUTs as IGLOO and iCE40 FPGAs. However, no evaluation board was available to us for our experiments, although Menta provided access to their design tools. As any technology node is supported in theory, lack of an IMD-suited node will certainly not be a problem. Practical implementations and power figures are not available to the best of our knowledge, so using the Menta eFPGA design tools and reporting energy and performance figures for the cases in Section 1.3 would give novel information on eFPGA feasibility in IMDs.

2.2. Cryptography for ultra-low-power hardware

Since the dawn of the computer age, modern cryptography has been used widely in securing digital communication. Three categories of cryptographic algorithms are distinguished here: that of symmetric block ciphers, hash functions and public-key cryptosystems. In this section, focus will be laid on lightweight cryptography as suited for IMDs. Within all categories, an industry standard is discussed and a lightweight version, if any, is selected as candidate for our experiments. Furthermore, complete security schemes specifically targeted at IMDs will be highlighted.

2.2.1. Symmetric block ciphers

A block cipher is a symmetric cryptographic algorithm that uses the same key for encrypting and decrypting data. Incoming data is split in fixed-size blocks, after which every block is encrypted with the same key as depicted in Figure 2.1. A similar procedure is repeated for decryption, with every block being individually decrypted. Block and key size can be an equal number of bits, however, this is not necessarily the case. The first publicly available block cipher is the Data Encryption Standard (DES) cipher, which is in a modified form (triple DES) still in use today for electronic banking. However, theoretical weaknesses have been demonstrated in the cipher, prompting its replacement. In 2000, the Rijndael algorithm [18] was chosen to replace DES in the Advanced Encryption Standard (AES). AES is nowadays one of the most used block ciphers and the de-facto industry standard in this category. Because it is seen as the reference benchmark among block ciphers, a lightweight version will be included in our experiments.

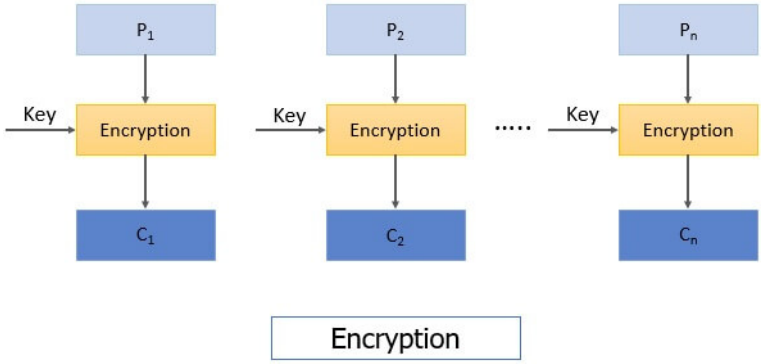


Figure 2.1: Schematic representation of encrypting data with a block cipher [32]

Although widely used, AES is not specifically optimized for operation on ultra-low-power hardware. As a result, numerous lightweight block ciphers have emerged over time to address this issue [33, 34, 35, 11, 19, 12]. KLEIN is a family of lightweight block ciphers that is geared towards application in RFID tags with tight resource constraints. However, its main focus is software implementations and although openly accessible hardware variants exist, no matching software and hardware implementations were available to us. The same holds for LED and PRESENT which are optimized for hardware implementations but were unavailable as well. A Midori repository has both software and hardware implementations available, but the software implementation is only available as Python code, needing conversion to C++ before we could include it in our experiments. According to [33], KATAN and KTANTAN are among the most hardware-efficient block ciphers requiring less than 1000 gate equivalents (GE). However, only SIMON had an openly accessible hardware and software implementation available, which is the main reason to select this cipher as lightweight block cipher representative in our experiments.

2.2.2. Hash functions

Hash functions have a different goal from block ciphers. Instead of mapping plaintext to ciphertext of equal length, hashing creates a fixed-size character string that is unique for each input of arbitrary length, minimizing the chance for a collision (two inputs generating the same hash). Hashing is useful for data integrity checking as a digital signature. A strong hashing function also makes it infeasible to reconstruct the input data if only the output hash is available.
Multiple hashing function families are designed over the years, with newer ones replacing older, broken, functions. The latest hashing function family that can be considered the industry standard is Keccak, integrated in the SHA-3 standard [36]. Similar to AES in the block cipher category, SHA-3 is not geared towards low-resource platforms. Specific lightweight hashing functions for these environments have been developed. Judging from literature, lightweight hashing functions are less in number than block ciphers, with the PHOTON family as only option with available hardware and software implementations. Hence, the choice for our experiments is easy, with PHOTON [20] being representative of lightweight hashing functions in our experiments.

2.2.3. Public-key cryptosystems

Following the publication of the Diffie-Hellman key exchange, cryptosystems which only required a secret key for decryption began to be published. Instead of having a single secret key that is pre-shared between parties to decrypt and encrypt, two keys are present in public-key systems: a public and private key. Encrypting a message can be done by anyone using a public key, which as the name indicates is known to everyone. Only for encryption, a secret key, paired to its public key, is needed. In Figure 2.2 the difference with symmetric cryptosystems is indicated. A most prominent example is the RSA algorithm, with many deduced variants.
As public-key cryptography tends to be more taxing on resources than symmetric cryptography and for lack of available lightweight implementations, this category is not elaborated further on in our experiments.

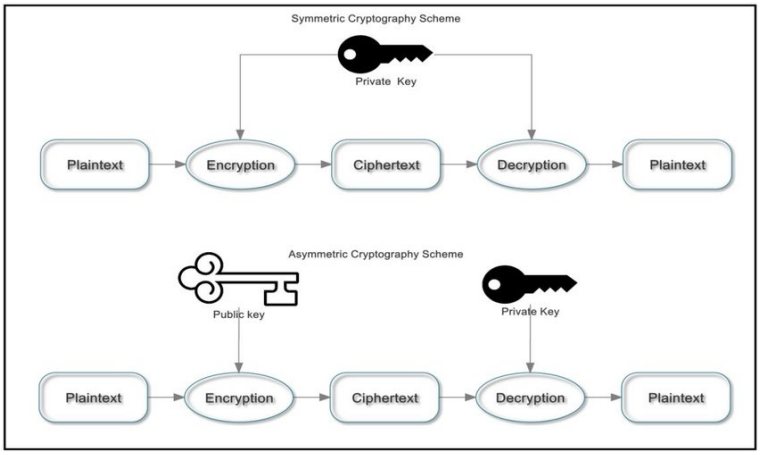


Figure 2.2: Symmetric and Asymmetric Cryptosystem Scheme [37]

2.2.4. Practical lightweight cryptography in literature

Looking now at cryptography specific to IMDs, one specific field of interest is to guard wireless channels against attackers. Increased power consumption and lack of computation power are among the reasons why security was often omitted. With ever increasing computational power, cryptographic security becomes more viable in IMDs. Table 2.3 lists implementations of several (lightweight) security schemes aimed for ultra-low-power operation. There is a noticeable difference between software and hardware-accelerated implementations of the same protocol. AES-128 for example, which is not designed to be a lightweight protocol, outperforms a software MISTY1 implementation in energy consumption while the latter one is designed to be lightweight. When speaking of hardware acceleration, one can make use of dedicated circuits (hard IPs or ASICs) or implement a scheme in an FPGA. The latter is interesting because of greater flexibility. Looking at slice occupation of FPGA implementations, it can be seen that normal symmetric ciphers fit well within low-power FPGAs with less than 400 slices (one slice consisting of 2x LUT4 and 2x FF). Even a public key scheme like Elliptic Curve Cryptography could possibly fit onto a bigger low-power FPGA. This shows that cryptographic schemes are excellent candidates for low-power FPGA implementation.
The only eFPGA entry [38] is a result of a collaboration between Menta and Secure-IC, who developed an embedded custom block (eCB) that can be integrated into a Menta eFPGA the same way as one would add a RAM or DSP block. This way, the LUT count needed for cryptography can be reduced drastically, even beyond the other listed implementations. However, due to unavailability, no evaluations are done on this implementations in this thesis.

Table 2.3: Hardware implementations of lightweight cryptography ciphers

Hardware implementations of lightweight cryptography ciphers						
Reference	**Protocol**	**HW accel.**	**Platform**	**Power (mW)**	**Energy (uJ)**	**Slices used**
[15]	SHA-1	Yes	Spartan3	6.6	132uJ	-
[16]	AES-128	No		-	50uJ	-
	AES-128	Yes	Cortex-M0+	-	2uJ	-
	SPECK	No		-	11uJ	-
	MISTY1	No		-	8uJ	-
[39]	Custom (PUFs)	No	Custom ASIC	1.18 – 11.37	∼2pJ/bit	-
[40]	MISTY1	No	Custom RISC	0.233	202uJ	-
[17]	PRESENT	Yes		-	-	117
	HIGHT	Yes		-	-	91
	Camellia	Yes	XC3S50-5	-	-	318
	TinyXTEA-3	Yes		-	-	254
	AES-128	Yes		-	-	393
[41]	ECC	Yes	AGLN250V2	-	-	2681 – 5126
[42]	SIMON	Yes	IBM 130nm ASIC library	-	-	1629 gates
[38]	AES-128	Yes	Menta eFPGA	-	-	embedded custom block (eCB)

2.3. Artificial Neural Networks and their application in low-power environments

Introduced in Section 1.3.2, artificial neural networks, ANNs for short, provide a promising approach for solving complex problems where conventional algorithms are unsuccessful. Tasks like object detection, image segmentation, speech recognition and other forms of classification can be carried out by neural networks with great success. But, ANNs are known for rather high hardware resource usage and power consumption. Hence, there has been an increased focus on making lightweight ANN models to be able to run them on Internet-of-Things devices [8, 9] and eventually, IMDs. In this section, general background on ANNs will be given and IMD-relevant networks will be highlighted to pick a candidate network for our experiments.

2.3.1. The artificial neuron

ANNs are modeled after the human brain and as such consist of a network of artificial neurons. The basic building block of an ANN is the artificial neuron, of which a representation is shown in Figure 2.3. Starting at the inputs, a neuron accepts external data that may be a part of the input data or outputs of other neurons. Taking the case of seizure prediction, input data could be brain signals that are captured by implanted electrodes. These input values are weighted, where the weights determine how important the contribution to the net input of each input is. Next, all weighted inputs are combined by summation and fed to the activation function. Being a non-linear function, it can be tuned to get the desired output for the designed problem. For different types of activation functions, see Figure 2.4. For binary classification, a sigmoid activation function is suited as it stabilizes at zero and one for low and high net input values. Likewise, the ReLU (Rectified Linear Unit) function is the most popular option for deep neural networks [43]. The resulting output of a neuron gives indication of the class of its input values and functions as pattern detection. Its output value can be passed to other neurons or used as network output.

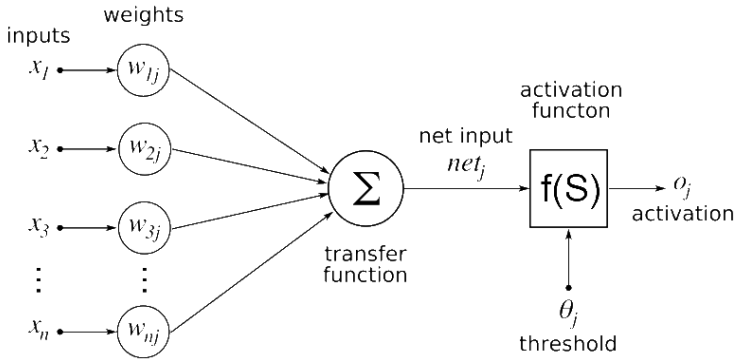


Figure 2.3: Schematic representation of an artificial neuron [44]

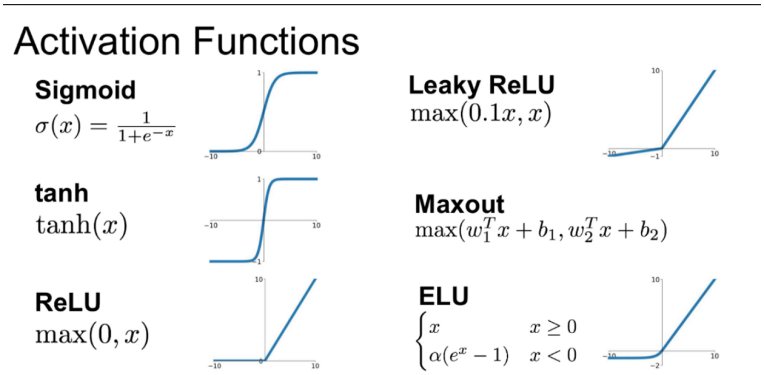


Figure 2.4: Commonly used activation functions [44]

2.3.2. ANN types

Combining multiple artificial neurons in a network enables more advanced pattern detection. Networks can vary from one to multiple layers of neurons. Layers that are always present are the input and output layer, with a number of optional hidden layers. A simple schematic is shown in Figure 2.5. These networks all have different properties that make them suited for different computational problems. Over time, many variants and classes have emerged. As a deep dive into the workings of neural networks is considered out of scope for this thesis, only a few examples that are used in low-power environments and assessed literature are highlighted, all based on the feedforward neural network topology. Some examples are:

1. **Feedforward Neural Network (FNN).**
   In this class of neural networks, data moves in one direction from input layer through any hidden layers to the output layer, no loops or feedback is present here. The basic building blocks of feedforward networks are neurons like described in Section 2.3.1. Every neuron in a layer has its output connected to all neurons of the next layer like in Figure 2.5. This layer topology is called a fully connected layer. An FNN is the simplest type of neural network, with many variants that add other layer types for faster or more efficient operation on specific problems. In this thesis, focus is put on this class of neural networks, as variants with simpler topology tend to have lower hardware resource usage and power consumption which are strong requirements for the ultra-low-power environment of IMDs.
2. **Convolutional Neural Network (CNN).**
   A CNN is a type of feedforward neural network that, in addition to fully connected layers with regular neurons, adds convolutional layers that have a filtering and flattening effect. Used frequently for 2D data like images, the convolutional layers create multiple convoluted (shifted) versions of the input, which are reduced by pooling layers. This process can be done multiple times in multiple layers until the 2D input image is completely flattened. After convolution and pooling, the flattened input is presented to one or more fully connected layers which perform the actual classification. An example is shown in Figure 2.6, where an input image is flattened by convolution and pooling layers before being classified as a digit between zero and nine. CNNs are used mostly for image classification, speech recognition and other two-dimensional problems, where they have shown superior performance. The neural network used in the experiments in this thesis is of the CNN class used for speech recognition, hence emphasis is put on this specific type of neural network.
3. **Binary Neural Network (BNN).**
   Although strictly speaking not a distinct type of neural network, a binary neural networks are aimed at greatly reducing hardware cost of ANNs. Its weights and activation functions are stored as binary values, instead of floating-point like the classical neural networks. An instant disadvantage of this change is the prospected drop in classification accuracy, partly due to the fact that weights cannot be precisely tuned. A major advantage is the reduction in needed memory to store weights and replacing floating-point computations with binary additions. Especially in the field of IMDs with

highly restricted hardware resources, BNNs could be providing the needed hardware reductions to make ANNs feasible.

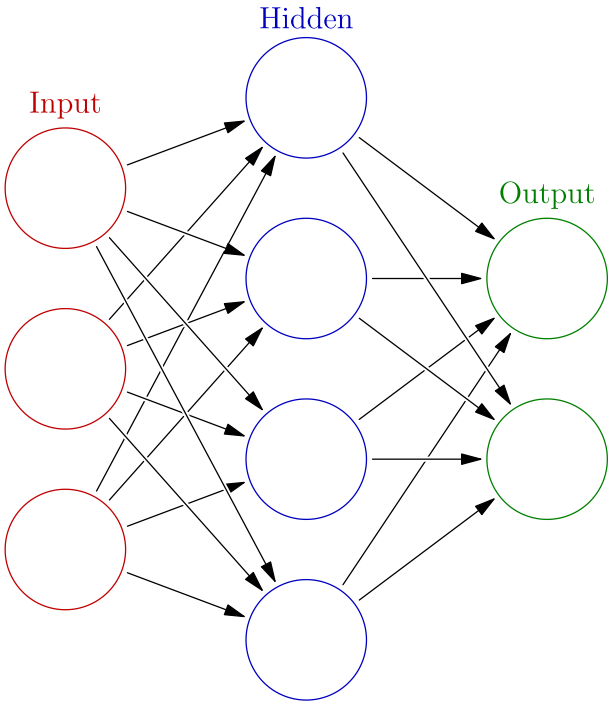


Figure 2.5: Feedforward neural network with one input, one output and one hidden layer. [45]

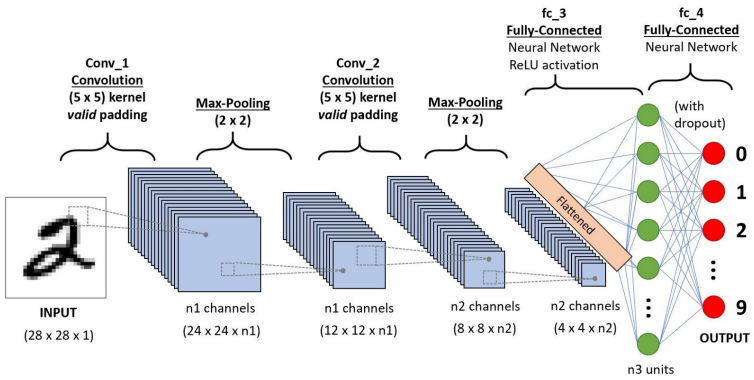


Figure 2.6: Steps in classifying an image with a CNN [46]

2.3.3. Network training and inference

After having designed the topology of a neural network, it cannot be used right away for its intended classification task. Two phases can be distinguished: training and inference, of which a visual representation is shown in Figure 2.7. First, the network needs to be trained, which is another word for letting the network learn to recognize patterns. A set of training data that has already been classified by hand is fed to the untrained network, after which the network output is compared with the 'real', correct, answer. If we take seizure prediction for example, a training data set could be a set of a large number of brain signals, measured just before and during a seizure, mixed with signals during normal brain operation. The output of the network can be a 'yes' or 'no' to the question of an input signal indicating

an upcoming seizure. Depending on the answer, the weights in the network are updated and a new input is evaluated. This process continues until the desired success rate is achieved.
With a trained network, inference can be done, which boils down to actually using the network with data outside of the training set. This step is considerably less computationally intensive than training and is feasible to run on low-powered hardware [23]. With all neural networks considered in this thesis, training is done offline (that is, on a powerful computer) while inference is online on the FPGA or CPU itself.

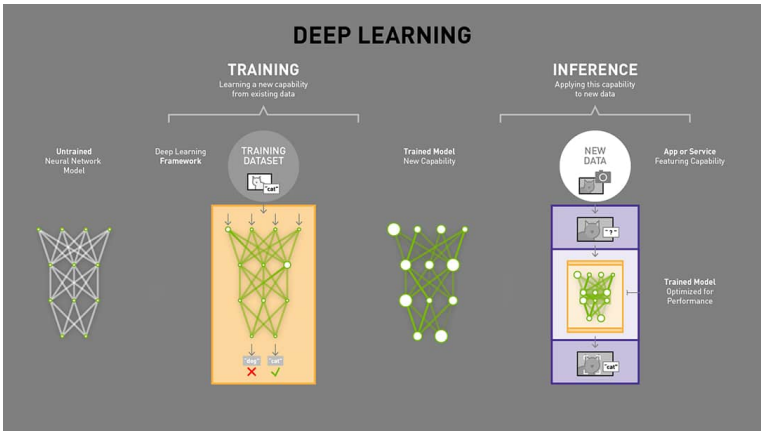


Figure 2.7: The training and inference phases of a neural network [47]

2.3.4. Artificial Neural Networks in IMDs

IMD-suitable neural networks in literature

As mentioned in Section 1.2.2, designing an IMD-suited neural network is considered out of scope for this thesis. Instead, literature was explored to gain insight in the state-of-the-art of tiny neural networks. The goal of this exploration is to find a small neural network that has the following characteristics:

1. Having a functional identical software and portable FPGA hardware implementation
2. Fitting in the available resources of our baseline FPGA, the Lattice iCE40UP5K [10]
3. Consuming less than 10mW when active on the FPGA
4. Clear documentation of the project with testbench included

First of all, the search was directed at seizure detection implementations as this application is well suited for ANNs. Solutions have been implemented on MCUs [21, 22]. A big name in microcontroller-powered ANNs is TinyML [23], but only targeted at MCUs and not FPGAs. However, as we are looking at an FPGA implementation for our experiments, these can only serve as baseline MCU data point comparison. As neural networks tend to be resource and power hungry, the majority of them are not suited for IMDs. Numerous FPGA neural networks can be found in literature which are too large in LUT count [48, 49, 50], exceeding 10k LEs and geared towards higher-performance application. Another obstacle is using vendor-specific IP like RAMs and DSPs, which makes an implementation not portable to another platform. Hard macros like RAMs and DSPs can be defined in generic VHDL or Verilog code, but designs tend to be optimized for a particular platform, using vendor-specific code. Most found implementations that are tiny enough to fit in the required resource range are targeted at the Xilinx Zynq-7000 platform and written in HLS code [51, 52, 53, 54] or for the older Cyclone II [55]. No seizure prediction implementations have been found, which remains an open problem.

Looking specifically for BNNs to increase the chance of finding a tiny NN, implementations were found that still required porting work [56] or were targeted at high-end FPGAs [57]. The best BNN candidate is the TiNNBiNN network [9], as that is targeted at the iCE40UP5K FPGA and thus would not require any porting work and automatically fulfills the resource requirement. However, in the duration of this thesis project, no source code was available to be able to use it for our experiments.

The most promising ANN that is fitted specifically for our baseline FPGA and fulfills all requirements but the first (no software implementation available), is the Lattice CNN Compact Accelerator IP [58]. Being a part of the Lattice SensAI stack and fitting in under 4000 LUT4 cells, this CNN could be a good candidate for implementation in an FPGA-equipped IMD. No comparison can be made with a functionally identical software implementation, but a general picture of feasibility of this CNN on an IMD is certainly possible. The smallest project where this CNN is integrated is the Key Phrase Detector [59] that listens for spoken key phrases and activates on a correct one in a simple yes/no fashion. Noting that this project has been designed by Lattice for their own FPGAs, it is certain that no porting work is needed to our baseline FPGA from the same vendor which is a substantial advantage over other ANNs. By lack of an ANN that fulfills all four requirements, this one is chosen for our experiments.

ANN to HDL mapping engines

Another way to obtain an ANN that fulfills the four requirements listed in this section is that of mapping engines, which convert a neural network description into C++ or HDL code. This way, both software and hardware implementations can be derived from the same ANN description, fulfilling the first two requirements. Picking the right network parameters can further fulfill the third requirement. The fourth requirement does not allow the use of undocumented repositories, of which there are plenty.

The first mapping engine example is SeeDot from Microsoft [60], which is geared towards generating FPGA and software implementations from a high-level mathematical network description. Furthermore, its focus is primarily on low-precision machine learning for IoT devices, which translates to suitability for resource-limited devices. Unfortunately, its repository is offline and therefore this mapping tool cannot be used anymore.

Next, EdgeML [61] is available what seems to be the successor of SeeDot. Like the latter, EdgeML is focused on IoT edge devices with little hardware and energy resources. Unlike SeeDot, this tool has an active software repository. However, the generated FPGA implementations are only targeted at Xilinx devices and will require extensive porting work to make them run on our baseline Lattice FPGA. As these tools were only discovered later in the duration of this thesis, no time could be afforded to port an implementation for use in our experiments.

Last, DL2HDL is a tool presented in [62] that maps ANNs described in PyTorch to HDL code. No equivalent software implementation is generated by this tool, but another PyTorch to C++ converter could be used for this. Again, due to the amount of time needed to get a design working and understood on our baseline FPGA, no use was made of this tool.

2.4. Wireless energy harvesting

To guard against battery depletion by adversaries, there is the option of running the hardware that is responsible for wireless communication on externally harvested power. Doing so to mitigate battery Denial-of-Service attacks has been named Zero-Power Defense [63]. Also, the battery life of an IMD can be extended by using harvested power. Energy harvesting can be done in different ways, via an inductive, acoustic or RF channel [64][65]. Table 2.4 shows several energy harvesting setups in literature. Most solutions use inductive power transfer (IPT) as this yields the most power transferred from a reasonably close range. A relatively high power transferred is preferable, as this looses the upper bound on IMD hardware power consumption. However, the use of capacitors can make up for a lower transferred power at the cost of latency (waiting for the capacitor to charge). From Table 2.4, it can be concluded that running an FPGA with several milliwatts of peak power will be very well possible on harvested energy. This matches the typical power draw of low-power FPGA cases listed in Table 2.1. In this research, the value of 6.15mW from [66] is taken as harvested power upper bound. Translating this information to our experiments, this bound is only needed for the security case in Section 1.3.1 as the FPGA case 2 in Section 1.3.2 is not used for communication. Instead, the implementation of Case 2 will not run on harvested power, but is limited by available energy in the IMD battery.

Table 2.4: Wireless energy harvesting in literature

Wireless energy harvesting in literature						
Reference	**Harvesting setup**	**Harvested power (mW)**	**Energy needed (uJ)**	**Energy used for:**	**Authentication time**	**Capacitor**
[15]	33mm coil pair	tens of mW	132	One-time challenge-response authentication & SHA-1 encryption	132ms	-
[13]	-	-	108.3	One run of IMD-fence protocol	15.7ms	-
[16]	miniature custom IPT circuit	6.15	35	Reader 32-bit command, IMD 64-bit response	10-500ms	10uF-460uF
[24]	915MHz RF setup	0.194	7.45	Entity authentication		
[64], [66]	miniature custom IPT circuit	6.15	20.07	Entity authentication	5.27-8.85ms	10uF
[67]	Thermoelectric Generator	0.065	-	AHRS and BLE 4.0 connectivity	-	
[4]	1.5-3GHz RF	mW range	-	BLE module	-	0.1pF

2.5. Conclusions

In this chapter, we gained background knowledge on the following four subjects: Ultra-low-power FPGA and eFPGA technology, cryptography for ultra-low-power hardware, artificial neural networks and their application in low-power environments, and wireless energy harvesting.

FPGAs have become a viable option for the ultra-low-power domain with recent technology developments. They have been proposed for IMDs in literature, however, no extensive energy analysis on FPGA-equipped IMDs has been done. To be able to answer our first and second research question and contribute to this knowledge gap in literature, a baseline FPGA (Lattice iCE40UP5K) and eFPGA technology is used in our to be conducted experiments.

In IMDs with wireless connectivity, securing its communication is of utmost importance, especially in life-critical applications. Most commercial security implementations run on MCUs in software, which makes all but the most basic primitives infeasible to run. Hardware accelerated blocks within MCUs have been used for a tremendous performance and energy gain, however, no fast alternative is available in the case the accelerated primitive is broken during the lifetime of an implanted IMD. FPGAs could combine the advantages of both CPUs and ASIC peripherals and as seen from literature, cryptographic schemes are excellent candidates for low-power FPGA implementation, resource and energy wise. As such, hardware-accelerated cryptography in IMDs is considered as first use case in this thesis.

No IMD-suitable hardware-accelerated neural network could be found that fulfills all of our four requirements in literature. Designing an efficient ANN on a low-power FPGA is a difficult task, judging from the lack of well-documented suitable implementations. Looking at mapping engines to generate ANNs that do fulfill all requirements, time was too limiting to be able to use one for our experiments. The best readily available ANN that has a well-documented FPGA implementation is the Lattice CNN Compact Accelerator IP. This IP is integrated in the Key Phrase Detector example project which will be taken as reference ANN for the second use case in our experiments.

Wireless energy harvesting was discussed with the goal to mitigate battery drain by adversaries. A typical continuous power draw was found to be 6.15mW from literature for the security primitives in Case 1 (Section 1.3.1), which will be used as upper bound in this research. The power limit of Case 2 (Section 1.3.2) will not be defined by this limit but rather indirectly by prospected battery life calculations.

3

Design

In Chapter 1, we defined our problem statement and four research questions derived from it. To answer our research questions, two use cases were introduced for which experiments are constructed in this chapter: an IMD-FPGA serving to secure wireless communication, and a functional unit integrating a hardware-accelerated CNN.

Having observed the technological trends of IMDs in Chapter 2, we have seen the need for IMD hardware taking up increasingly advanced signal processing tasks, like pattern detection to help preventing seizures. Likewise, the addition of wireless communication to IMDs calls for securing that communication channel with cryptography. However, extra algorithm processing leads to increasing CPU time and energy consumption on already heavily power-constrained MCUs present in modern IMDs.

Before going to the experiments we discussed four topics relevant to our two use cases in Chapter 2: low-power FPGAs, cryptography, artificial neural networks and wireless energy harvesting, all in low-power environments. With this background, a baseline FPGA and four algorithms were selected for our experiments. Furthermore, an upper bound of 6.15mW was set for Case 1 to mitigate battery draw by adversaries.

In this chapter, our two use cases will be converted into a benchmark suite that consists of FPGA implementations around three security primitives and one artificial neural network. The goal of this benchmark suite is to be able to evaluate it on energy, area and resource usage to see under what conditions said implementations are feasible in IMDs. For a discussion on these results, see Chapter 4.

In Section 3.1, an overview of a typical modern IMD is given and the part of interest for our design is indicated. In Section 3.2 a technical overview of the selected security primitives and artificial neural network is given. Next, an in-depth description of the from-scratch designed interfaces for the three selected security primitives is presented in Section 3.3, together with the assembly of the interfaces to all security primitives. Third, the different MCU, FPGA and eFPGA platforms which run the selected algorithms are shown in Section 3.4. Further explanation on the eFPGA topology and tools used can be found in Section 3.5. Necessary modifications to prepare the designed FPGA implementations of our benchmark suite are described in Section 3.6. Conclusions on this chapter can be found in Section 3.7.

3.1. Design overview and experiment goals

To understand what IMD part is modeled in our designs, we have to know the basic topology of an IMD. In Figure 3.1 a block diagram of a typical modern IMD architecture is depicted. Separate sensor, actuator and memory parts are connected to the processor via a high-bandwidth interconnect. The processor is the master device which runs the main algorithms and control loops. Communication to the outside world happens through a transceiver which is optimized for low power consumption.

In this chapter, we will zoom in on the *Extra Processing* block, which is an auxiliary hardware device that can accelerate parallelizable workloads. Normally, hardware acceleration is performed through special ASIC blocks within MCUs but, as noted in Section 2.1.1, implementing this block on a low-

power FPGA suitable for IMDs is a novelty that is largely unexplored. The design presented in this chapter is a benchmark suite that will make quantitative evaluation of FPGAs in IMDs possible. The suite consists of security primitives and a CNN implementation that will run on the *Extra Processing* hardware block, which is an FPGA in our case. The design presented in this chapter is not that of a complete IMD architecture, but that of the *Extra Processing block* shown in Figure 3.1. This block is then implemented on FPGA fabric and compared with functional identical software algorithms running on reference IMD processors. Added to the *Extra Processing* block, bus interfaces are designed from scratch to be able to connect it without much effort to an existing IMD architecture. In Figure 3.1, these interfaces are represented by the dashed gray arrow. With our designs, we will be able to perform power measurements and make accurate prediction on the impact on IMD battery life when adding them in the *Extra Processing* block.

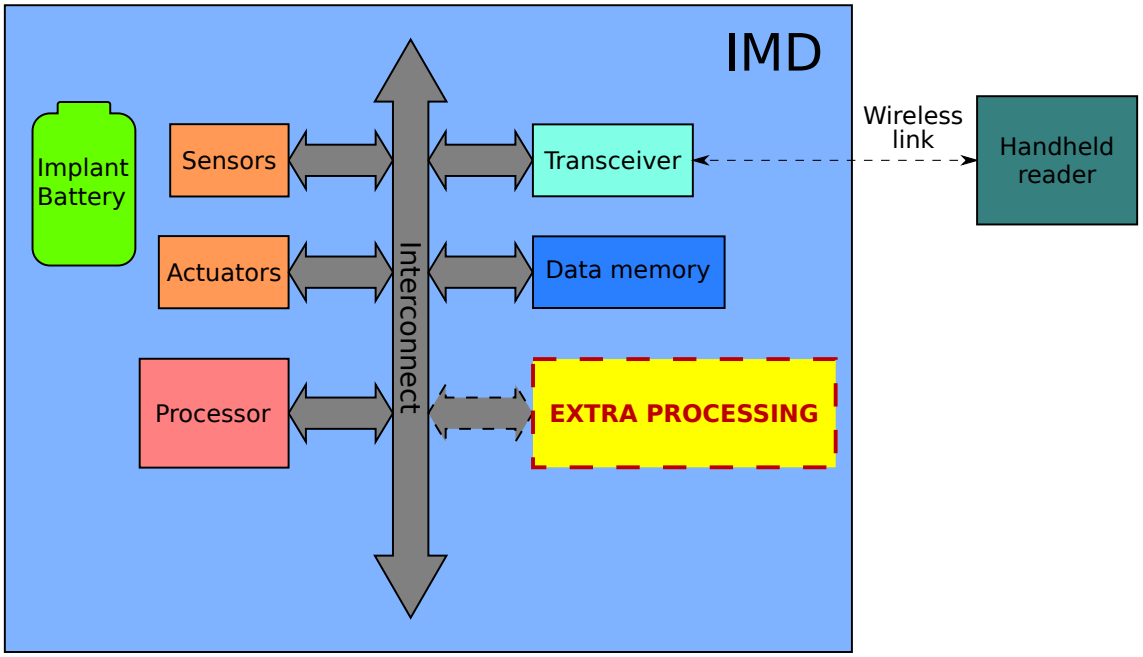


Figure 3.1: Top-level schematic view of the hardware of a modern IMD

3.2. Selected algorithms

For the first class of secure wireless communication, we selected three algorithms to represent a broad spectrum of IMD-ready security primitive implementations. FPGA implementations are to be compared to traditional implementations running on MCUs, meaning that C/C++ and HDL implementations of every algorithm need to be available. The second class, that of machine learning, has existing implementations on tiny FPGAs [8]. However, an analysis considering the environment and energy constraints of IMDs has never been done for tiny FPGAs to the best of our knowledge. Employing artificial neural networks on IMD-suitable FPGAs is a novelty that is explored in this thesis. All selected algorithms are listed in Table 3.1 and discussed subsequently in the subsections below. The FPGA implementations will be discussed in detail, whereas the C/C++ codebases are drawn from previous research and will not be further elaborated on here.

3.2.1. AES-128

Advanced Encryption Standard (AES) currently is one of the most used block ciphers [18]. Proposed in 2001, AES is a family of ciphers with a fixed block size of 128 bits and three different key lengths of 128,192 and 256 bits. To keep potential hardware resource usage to a minimum, the least complex variant of AES is selected: AES-128, which has a key and block size of 128 bits.

Table 3.1: Selected algorithms

Algorithm	Type	Block/hash size (bits)	Key size (bits)	Reason for selection
AES-128	Block cipher	128	128	Widely used, simplest version (128-bit key). Included to compare CPU, ASIC & FPGA implementations.
SIMON-64/128	Block cipher	64	128	Lightweight alternative to AES. Included to compare algorithm complexity.
PHOTON-128	Hashing algorithm	128	-	Lightweight hashing alternative to SHA-3 (which is too costly for IMDs)
Key Phrase Detector	Compact CNN	-	-	Good representative of ANNs for IMDs (offline training, online inferencing).

Table 3.2: AES-128 ports description

Port name	Direction	Width (bits)	Data type	Description
clk	Input	1	clock	System clock.
data_i	Input	128	data vector	Input plaintext
decrypt_i	Input	1	toggle	Indicates if encryption or decryption is going to take place.
key_i	Input	128	data vector	Key used for encryption and decryption.
load_i	Input	1	positive-edge trigger	Signals the start of a one-block operation.
reset	Input	1	toggle	Active low reset.
data_o	Output	128	data vector	Output ciphertext.
ready_o	Output	1	positive-edge trigger	Signals stable output data available.

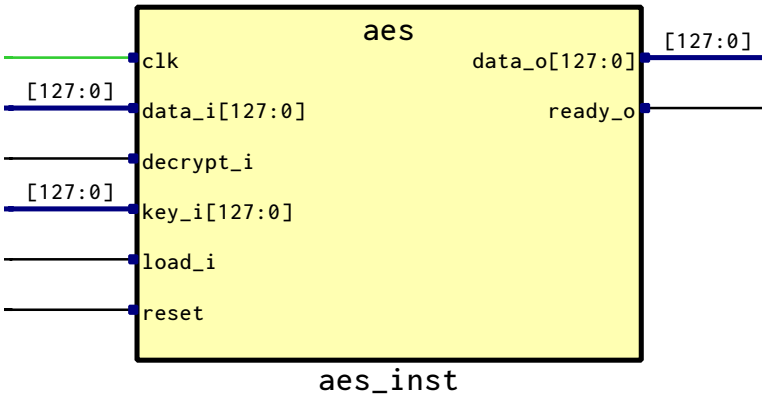


Figure 3.2: Top-level schematic view of AES-128 IP core

AES-128 FPGA IP Core

As a base, an AES-128 IP core from OpenCores.org was selected that is optimized for low resource usage [68]. Originally written in SystemC, the code has been converted to Verilog by the author, which is used as the basis for the design present in this project. The AES-128 core fits in less than 2600 LUTs and does not use hard macros, which generally improves its portability between FPGA architectures. A top-level block schematic of this core is shown in Figure 3.2. A description of all ports is found in

Table 3.2.
As can be noted from Table 3.2, input and output data is presented as a 128-bit vector, making it easy to design an interface for this block, as the data only needs to be stable during operation. No advanced clocking mechanism is needed to feed input data to the AES block and the same goes for output data that can be read in one clock cycle. $decrypt$ is set before operation to indicate the kind of operation to take place. Decryption is, simply put, the inversion of the encryption operation and therefore can make use of the same hardware as encryption. However, only the encryption mode is used in the experiments as the SIMON IP core used [69] only implements encryption. The key can be changed for every 128-bit input but has to be stable while the AES-128 IP core is running.

3.2.2. SIMON-64/128

Designed by the NSA and published in 2013, SIMON was proposed as a lightweight block cipher optimized for performance in hardware implementations [19]. Its sister algorithm, SPECK [19], is optimized for software implementations. As an FPGA implementation of SIMON is going to be compared to its software implementation, it would be a logical step to compare hardware SIMON against software SPECK. But since no hardware variant is available of SPECK and both hardware and software are available of SIMON, it is chosen to compare only the SIMON implementations to each other to limit the number of variables.
SIMON can be seen as a lightweight alternative to AES, which could be promising for resource-constrained devices like IMDs. The main reason for including SIMON in the experiments is to compare within the group of block ciphers, what advantages a lightweight block cipher like SIMON has compared to the standard of AES. Just like AES, SIMON is a family of ciphers with different block and key sizes, ranging from 32/64 to 128/256 (block/key). As only the 64/128 variant was available as VHDL implementation, this configuration was selected for the experiments.

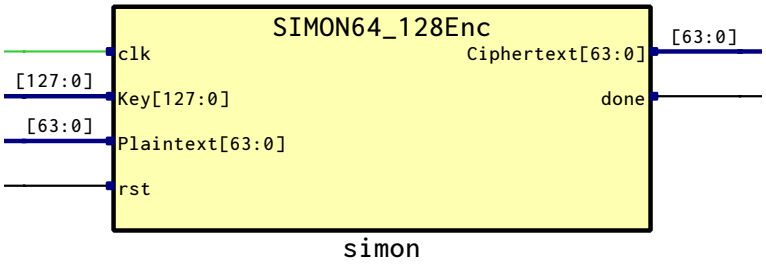


Figure 3.3: Top-level schematic view of SIMON-64/128 IP core

Table 3.3: SIMON-64/128 ports description

Port name	Direction	Width (bits)	Data type	Description
clk	Input	1	clock	System clock.
Key	Input	128	data vector	Key used for encryption and decryption.
Plaintext	Input	64	data vector	Input plaintext.
rst	Input	1	toggle	Active high reset.
Ciphertext	Output	64	data vector	Output ciphertext.
done	Output	1	positive-edge trigger	Signals stable output data available.

SIMON-64/128 IP Core
Both software and hardware implementations of SIMON are available, making a fair comparison feasible. From previous research considering fault injection in cryptographic hardware of lightweight ciphers [42], its SIMON-64/128 hardware implementation is borrowed for this thesis [69]. Being a lightweight cipher, the SIMON core is expected to consume significantly less resources than a 'standard' block cipher like AES. This assumption is confirmed later in Section 4.1.1. Compared to AES, this SIMON

implementation accepts plaintext blocks of half the size (64 bits), but uses a key of equivalent length. Therefore, the differences in resource usage between AES and SIMON will not only be due to the cipher itself, but also to the fact that this SIMON IP core accepts smaller plaintext blocks. As will be shown however in Sections 4.2.1 and 4.2.2, the number of rounds (and therefore the number of clock cycles) will in any case be significantly lower than AES, having major energy benefits over the latter.
As with AES, plaintext and key are presented as vectors to the inputs of the IP core. A port list is presented in Table 3.3 and the top-level RTL view in Figure 3.3. As stated in Section 3.2.1, no decryption option is provided, so only encryption performances are going to be compared. The input interface is simplified compared to the AES IP core, with the rst signal also acting as load trigger. All other ports are analogous to the AES IP core.

3.2.3. PHOTON-128

Secure hashing in an IMD might ultimately be possible with PHOTON [20], a lightweight hashing algorithm that borrows mechanisms from AES for its permutation function and relies on the sponge functions framework [70]. As insecure or broken primitives are not considered here, hashing algorithms like MD5 and SHA-1 are not implemented. 'Real' SHA-2 and SHA-3 IP cores cannot fit in a tiny FPGA of around 5k LUTs, generally speaking. A relatively small implementation of SHA-3 was tried [71], but we were unable to port it successfully to the Lattice FPGA platform. Hence, PHOTON was chosen as its authors provide a software and hardware implementation, its lightweight focus and solid basis on modern cryptography.

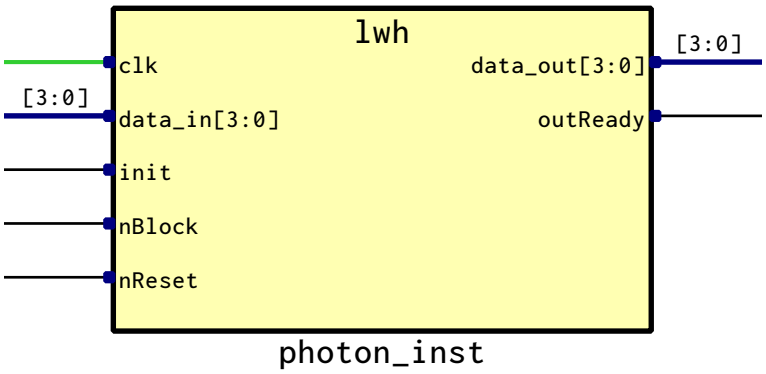


Figure 3.4: Top-level schematic view of PHOTON-128 IP core

Table 3.4: PHOTON-128 ports description

Port name	Direction	Width (bits)	Data type	Description
clk	Input	1	clock	System clock.
data_in	Input	4	serial nibble stream	Input message to be hashed.
init	Input	1	Toggle	Signals block operation, active high.
nBlock	Input	1	Toggle	Signals data input, active high.
nReset	Input	1	Toggle	Active low reset.
data_out	Output	4	serial nibble stream	Output hash.
outReady	Output	1	Toggle	Signals stable output data available.

PHOTON-128 FPGA IP Core

Unlike the AES and SIMON IP cores seen before, the PHOTON core has a data port that does not accept the input message in parallel at once. Instead, the input is clocked in over multiple cycles. For a port list and top-level view, see Table 3.4 and Figure 3.4. This way of feeding input data makes sense due to the nature of hashing algorithms: they are designed to create a fixed length hash output, 128 bits in this case, of an input message of arbitrary length that is not necessarily split up in predefined blocks. In this experiment, a single input message of also 128 bits is used as sample data, just like with the AES IP core. It would not be fair to compare a single-block AES operation to PHOTON with a very large digest message and conclude that PHOTON is slower! For the actual test setup, refer to Section 3.6.

The ports description and schematic view of PHOTON-128 are shown in Table 3.4 and Figure 3.4 respectively. The reset procedure is different from a simple level-active signal and an interface needs to translate the message to be digested into a serial stream of half-bytes or nibbles. For the designed interface, see Section 3.3.1 where the correct reset procedure and data handling is presented.

Removing clock gates in IP core

The IP core as provided by the author was working correctly, however some modifications were required. The design makes use of an array of linear-feedback shift registers (LFSRs) of which the shifting behavior is controlled by clock gating separate counting registers if no data shift is required. The problem with this is that the Menta synthesis tool (discussed in Section 3.5.1) routes every internally generated clock as a separate clock signal, not using the eFPGA routing network. This is important to ensure minimum clock skew and delay on the clock trees. For the PHOTON IP core, this results in over 30 separate clock domains. As the maximum number of clocks in Menta eFPGA fabric is 8, clock gates had to be removed and the clock gate enable signals rewired to the enable flag of the respective registers. In many synchronous designs, this is the preferred practice. The modification of such a counting register is seen in Figures 3.5 and 3.6. Note that the clock signal is not the same in both configurations: in the original it is externally gated whereas in the modified version, clk is the system clock which is always active. The added enable signal acts now as activation for the counter register inside. With this modification, the design is finally seen as having a single clock domain and made synthesizable on the Menta platform.

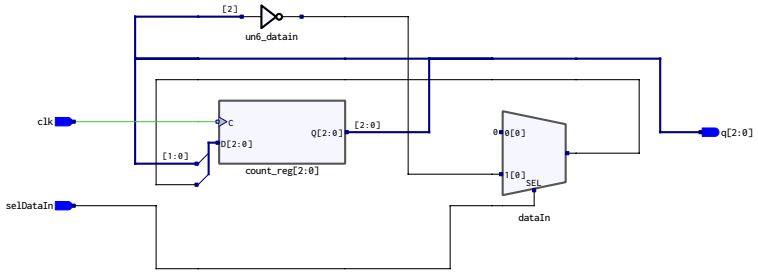


Figure 3.5: Original configuration with externally gated clock signal

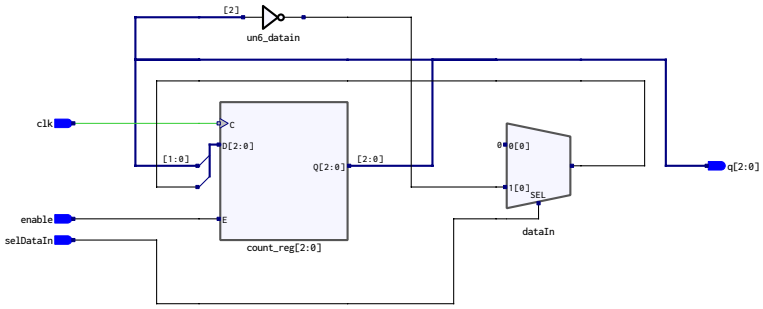


Figure 3.6: Modified configuration with system clock signal

3.2.4. Lattice CNN Compact Accelerator

FPGA implementations of neural networks that are suited for tiny FPGAs are hard to get by as discovered in Section 2.3.4, and as a result, the selected algorithm is a proprietary 6-layer Convolutional Neural Network (CNN) provided by Lattice for their FPGAs. A reference implementation using this CNN for key phrase speech detection was selected for our experiments [59]. This was the only available artificial neural network that:

- Fits in a tiny FPGA (less than 5,000 LUTs)
- Is well-documented
- has usable example implementations

Unfortunately, no software implementation to compare with a reference MCU platform exists as only a Lattice specific hardware implementation is available. Therefore, estimations will be made about the execution time and energy consumption on other platforms that are considered in this thesis, namely the MCU and eFPGA platforms. See Section 3.4.

Key Phrase Detector project with Lattice SensAI

The reference project used that contains the CNN, is a key phrase detector implemented on the Lattice iCE40UP5K FPGA [10]. Because of the amount of time required to study and implement a reference project with this CNN, this reference project was chosen instead. This implementation can distinguish a maximum of four key phrases at an evaluation speed of 40 phrases per second from received microphone input.

At the core of the detector is the 6-layer tiny convolutional neural network that is of particular interest. This neural network, NN for short is trained offline with common available tools like TensorFlow [72], Caffe [73] or Keras [74]. After a model is trained with reference data, a Lattice-specific tool, NN Compiler, translates the trained model into correctly quantized weights and instructions. In parallel, regular FPGA toolflow is followed to design the logic around the CNN. These two parts are combined in one bitstream to realize a Lattice CNN-equipped project. The complete toolflow is called SensAI and a graphical description is shown in Figure 3.7. For this thesis, a ready-to-go FPGA bitstream was used to directly employ the project on the FPGA. Therefore, no interface logic was designed like for the three security primitive IP cores in Section 3.3. Extensive explanations on the inner workings of ANNs in general is out of scope for this thesis. For more explanation on the Key Phrase Detector project, the curious reader can consult the manual [59].

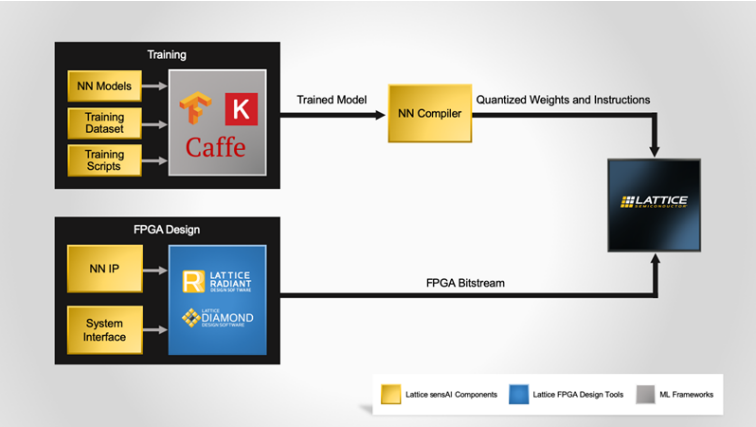


Figure 3.7: Lattice SensAI toolflow [75]

3.3. Selected interfaces

Since a crypto core needs to interface with its host processor in our case, two communication interfaces were considered and attached to the selected cryptographic block in separate implementations. These interfaces accommodate the needs of two different environments: firstly that of inter-chip communication, which can go through PCB traces or longer cables. Such an environment places limits on the

number of interconnect wires and data throughput. Multiple standards have been developed for this scenario like SPI, i2C and UART. The last one is chosen in this thesis to attach to all three crypto cores for its simplicity. Secondly, the environment of on-die communication is considered. Multiple protocols have been developed under the AMBA umbrella like APB, AHB and AXI with their different versions [76]. The version which has the smallest footprint and still provides enough functionality is AHB-Lite, also selected in this project. As the main focus is on the functional IP cores themselves, the interfaces are solely added to create a realistic resource usage picture. However, substantially more design work has been put in interfaces compared to the IP cores, as the first have been designed mostly from scratch where the latter were readily available IP cores.

3.3.1. Off-die: UART

A universal asynchronous receiver/transmitter is a digital device used for asynchronous bidirectional communication. Data is transferred in sequential fashion at a previously agreed data rate, breaking up the data in frames. A start and stop bit indicate the start and end of a frame. In most cases and also in this project, a frame consists of eight bits forming a byte, like in Figure 3.8.

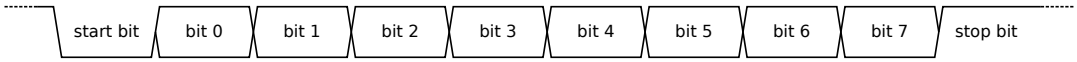


Figure 3.8: Diagram showing UART timing [77]

The goal of the UART interface is to abstract away the ports of the connected IP core and provide a TX and RX wire for incoming and outgoing data. At the core of every serial interface that will be visited in the following sections, is a simple UART with the ports shown in Figure 3.9 [78].

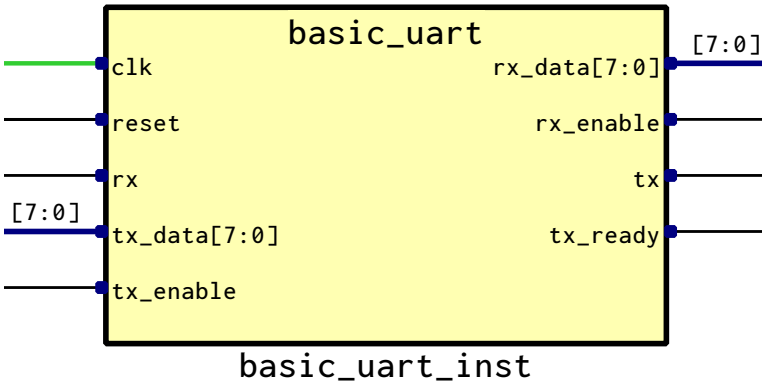


Figure 3.9: Top-level schematic view of UART module used

Operating the UART block is simple. For data to be transmitted, tx_data has to be held stable and tx_enable pulled high to start a transmission. During transmission, tx_ready is held low to indicate that the UART is busy. Once all bits have been clocked out of the tx port, tx_ready will pulled high again. The same procedure applies to the RX side, but all steps in reverse.
This UART has no TX/RX data buffers, which is a conscious design choice to reduce the total resource footprint. To see why this is not a limitation, note that the UART can never transfer data faster than an IP core can encrypt/decrypt incoming data, hence no need for an input buffer. The same goes for the output, which can always be read faster than the UART transfer speed. Timing complexity on the host processor side will slightly increase as not all data can be fed to the crypto core at once, but this can easily be mitigated in software.

AES + UART

Interfacing the AES-128 IP core as described in Section 3.2.1 boils down to the atomic operation of converting sixteen UART bytes to a 128-bit input vector. The same goes for the TX side: a freshly encrypted/decrypted block of data needs to be byte-serialized and presented to the $basic\_uart$ block. Interfacing between the basic UART block and AES IP core was realized by creating two blocks: RX and TX interface, of which the source code can be viewed in Appendix B.1.1. With state sequencing in the RX interface, it is ensured that sixteen consecutive bytes are captured from the RX port to construct a 128-bit input for the AES IP core. Similarly, the TX interface acts as a 128-to-8-bit converter, presenting output data byte sequential to the UART block. A complete graphical overview of the UART interface for AES is given in Figure A.1b. This design will serve as the basis for the other two designs presented in Section 3.3.1 and Section 3.3.1. A small block named ext_busy can also be seen in Figure A.1b. No extra functionality is added here, as this block only converts internal done and load signals to a single level-sensitive signal that indicates if the external core, in this case AES, is busy. This is useful for debugging purposes, as well as any external device that would monitor the status of the AES + UART block.

Furthermore, not only input ciphertext/plaintext can be fed though the RX line of this UART interface, but updates of the key can also be done. To make this possible, a simple multiplexer was put at the data output of the complete UART interface seen in Figure A.1b. For placement of all blocks in the top-level design, see Figure A.1a. For obvious security reasons, the key should be write-only. Next, the key needs to be held in a register to allow the input plaintext/ciphertext to change while the key stays static. In Figure 3.10 the resulting internals of the Key-or-Text (KorT for short) are shown.

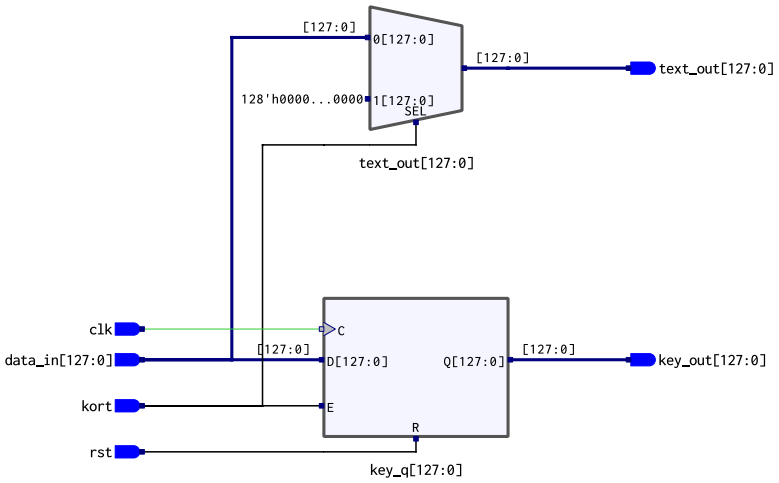


Figure 3.10: Key-or-Text logic for AES

SIMON + UART

Similar to interfacing the AES-128 IP core, the SIMON IP core needs a block that converts an incoming serial UART data signal to parallel text and key vectors. The main difference with AES lies in the halving of the block size, from 128 to 64 bits. These changes are reflected in the source code in Appendix B.1.2. For the RX interface, this means that eight consecutive bytes are captured from the UART rx_data port, which are converted to a 64-bit vector. In a similar fashion, the TX interface converts a 64-bit vector to eight sequential bytes, which are again converted to a bit-serial TX signal by the basic UART block. The schematic of the UART interface for SIMON can be seen in Figure A.2b.

Looking at the ext_busy block, the attentive reader could have noticed the $load\_in$ input is now coming externally instead of being driven by the UART interface of Figure A.2b. The reason for this change is that the reset signal of the SIMON block is shared with loading the crypto engine. There is no additional load signal, but the SIMON IP core has to be held in reset once input plaintext has been constructed and is stable at the input.

Regarding the selection to write key or plaintext, the simple multiplexer structure of Figure 3.10 was expanded with a simple state machine to ensure that a 128-bit key write was done by taking two consecutive 64-bit vectors coming from the UART interface. A plaintext write is done by simply passing the 64-bit vector from the UART interface to the $data\_in$ port of the SIMON block in Figure 3.3. Going up one level, the SIMON IP core with attached UART interface is shown in a top-level schematic in Figure A.2a.

PHOTON + UART

Compared to the AES and SIMON interfaces described in Section 3.3.1, the PHOTON UART interface seems to have the same blocks judging from Figure A.3b, except for the data port sizes which are now only four bits wide. However, the internals of the RX and TX interfaces are substantially different, which can be verified by viewing the source code in Appendix B.1.3. Unlike AES and SIMON, the PHOTON IP core has data input and output ports which do not present or read all data bits in the same clock cycle. The reason is that an input message of a hashing block can be of arbitrary size, unlike the AES and SIMON block ciphers shown before, which have fixed input and output block sizes. Data is instead clocked in nibble-wise serially, so four bits at a time.

Looking at the RX interface in Figure A.3b, the main bottleneck for presenting data to lwh, the PHOTON IP core, is the UART baud rate. Knowing that this baud rate is significantly lower than the system clock speed, the interface waits for one incoming byte from $basic\_uart$ and stores it in an internal register in two clock cycles following. This repeats until the desired input message length is reached, which is in this case hardcoded in the RX interface at 36 nibbles. The internal PHOTON state matrix consists of 36 nibbles of which 32 are the 128-bit input message. When the nibble threshold is reached, lwh is put in 'data input mode' and all stored nibbles, 36 in this case, are clocked in sequentially. As stated before in Section 3.2.3, the reset procedure is more complicated than a simple single-bit signal. Instead, three signals, namely $nReset$, $nBlock$ and $init$ control the reset, data loading and reading from the PHOTON IP core. Similar to the RX side, the TX interface gathers all lwh-output nibbles sequentially and stores them in a 128-bit register. Once the register is filled, data is sent to $basic\_uart$, one byte at a time, waiting to transmit the next byte until $basic\_uart$ has transmitted the current byte out of its TX UART port. No matter how large the input message was, the output will always be sixteen bytes or 128 bits.

3.3.2. On-die: AHB-Lite

In the scenario of an eFPGA integrated in an MCU on the same die, communication bandwidth and latency specifications are greatly improved compared to off-die communication. As stated in Section 3.3, the AHB-Lite protocol was chosen for its suitability to resource-constrained platforms. An AHB-Lite interface was designed from scratch and adapted slightly per crypto IP core, to abstract away the interfacing details from the AHB-Lite bus.

AHB-Lite general overview

AMBA AHB-Lite is a bus interface that supports a single bus master and provides high-bandwidth operation [79]. This master is typically a host CPU that interfaces with one or multiple slaves which can be memory devices, external interfaces of any sort and high-bandwidth peripherals. A schematic overview is shown in Figure 3.11. In the case of this design, the attached peripheral will be one of the three crypto IP cores disscussed in Section 3.2. Hence, an AHB-Lite slave interface is designed and presented, whereas the AHB-Lite master is not implemented. The reason is that only the peripheral, a crypto IP core in this case, is implemented in FPGA fabric. It is assumed that the host CPU will already have a master AHB-Lite interface integrated. A master interface initiates operations and has added control logic compared to a slave interface, making its resource usage higher. Figure 3.12a shows the master interface with ports. Figure 3.12b shows the slave interface. If there is only a single slave, the two interfaces can connect directly to each other, eliminating the need for a multiplexer and decoder in between.

One of the main contributors to the performance of AHB-Lite is the pipelining of addressing and data phases. When initiating a transfer, the master first sends out an address on the $HADDR$ bus. The next clock cycle, data is transferred over $HWDATA$ or $HRDATA$, depending on the operation being read or write. During data transfer, the address for the next data transfer is already put on $HADDR$. In Figure 3.13 this behavior can be seen, where the address phase of block B is occurring simultaneously with the data phase of block A. In the next sections, the specific modifications that make all three AHB-

Lite interfaces different will be outlined and design considerations shown. For source codes, refer to Appendices B.2.1 to B.2.3.

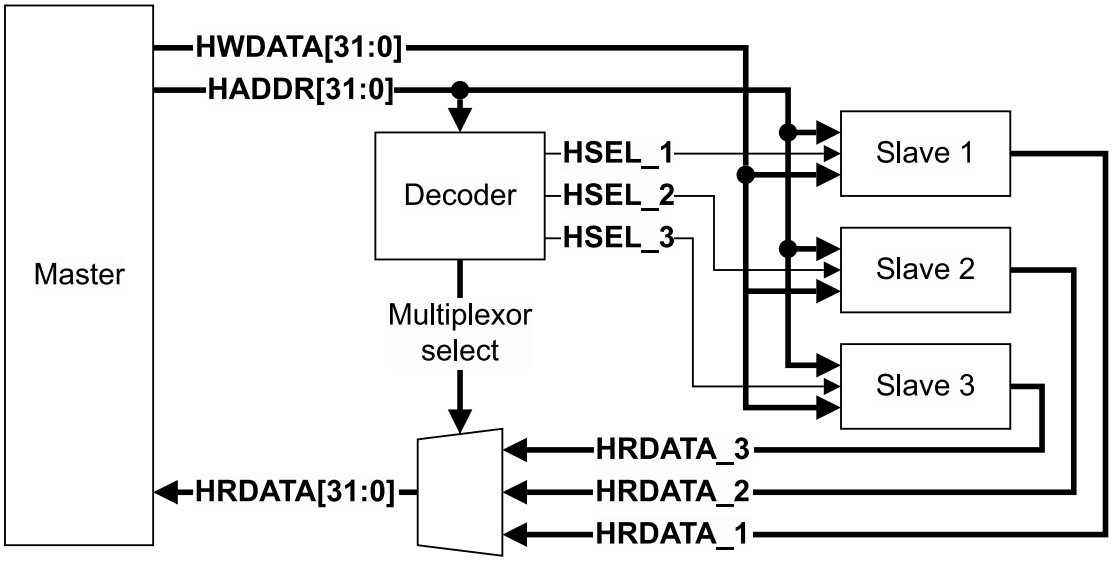
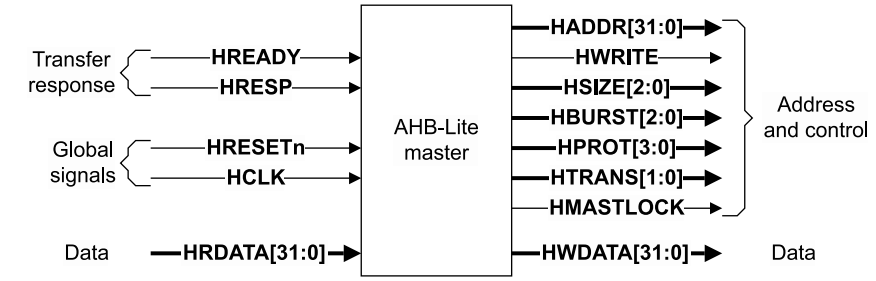
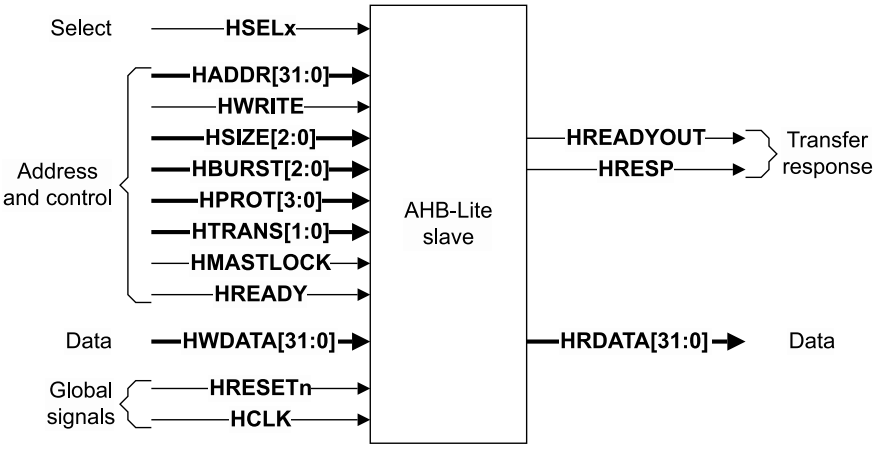


Figure 3.11: AHB-Lite top-level overview [79]



(a) Master interface



(b) Slave interface

Figure 3.12: AHB-Lite interfaces [79]

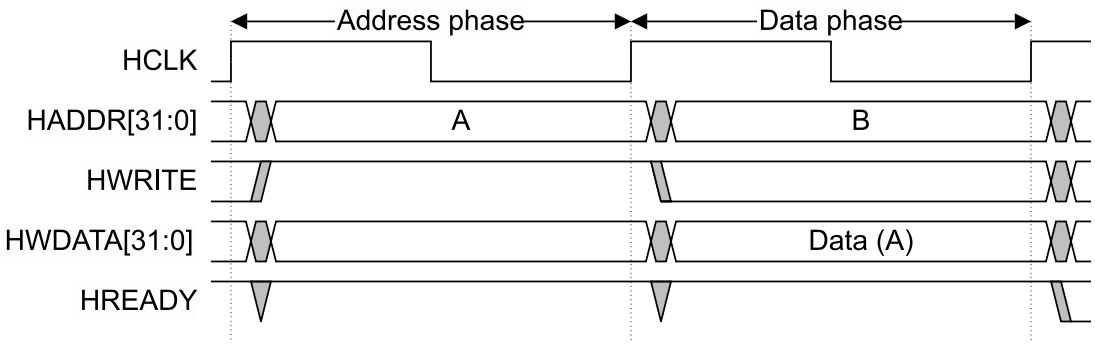


Figure 3.13: AHB-Lite pipelining of address and data phases [79]

AES + AHB-Lite

Just like the UART interface for AES in Section 3.3.1, the AHB-Lite slave interface for AES will be the basis on which the SIMON and PHOTON interfaces will be built on. Unlike the UART interface, the AHB-Lite slave interface has no basic premade building block inside. Instead, it was designed from scratch based on the protocol definition [79]. In Figure 3.14, a diagram of the simplified FSM created is shown. Five states can be distinguished with two intermediate states ($DECRYPT$ and $ENCRYPT$), which are sequenced automatically when going from $IDLE$ to $WRITETEXT$. The starting state is $IDLE$, where the FSM listens to valid values of $HADDR$, $HWRITE$, $HBURST$ and $HTRANS$. These signals determine what state is entered in the next clock cycle. In $WRITETEXT$, input plaintext or ciphertext, depending on whether $DECRYPT$ or $ENCRYPT$ was passed first, is written to the 128-bit register that holds one data block of input for the AES IP core. This is done in four steps (or a burst of four beats to stick to AHB terminology), as the AHB-Lite data line width is 32-bits to conform with common 32-bit CPU architectures. Reading the output value of the AES IP core and presenting it to the outside AHB bus is done in $READTEXT$, in a burst of four 32-bit beats. $READSTATUS$ results in two bits of information: whether the AES block is in decryption (1) or encryption (0) mode and if the block is busy (1) or idle (0). Most probably the least used state is $WRITEKEY$: having the same steps as $WRITETEXT$, but writing to the key instead of text register. The distinct $WRITETEXT$ and $WRITEKEY$ states also replace the functionality found in the Key-or-Text (KorT) block found in the UART interfaces in Section 3.3.1.

Attaching the freshly designed AHB-Lite interface to its AES IP core, the RTL top-level overview is shown in Figure 3.15. The top-level ports will be the same for all implementations with AHB-Lite in this project: 32-bit address and data, and the minimum necessary control signals.

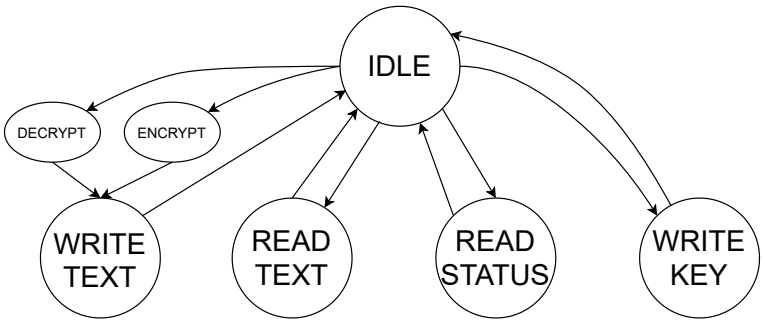


Figure 3.14: Simplified FSM describing the AHB-Lite interface for AES

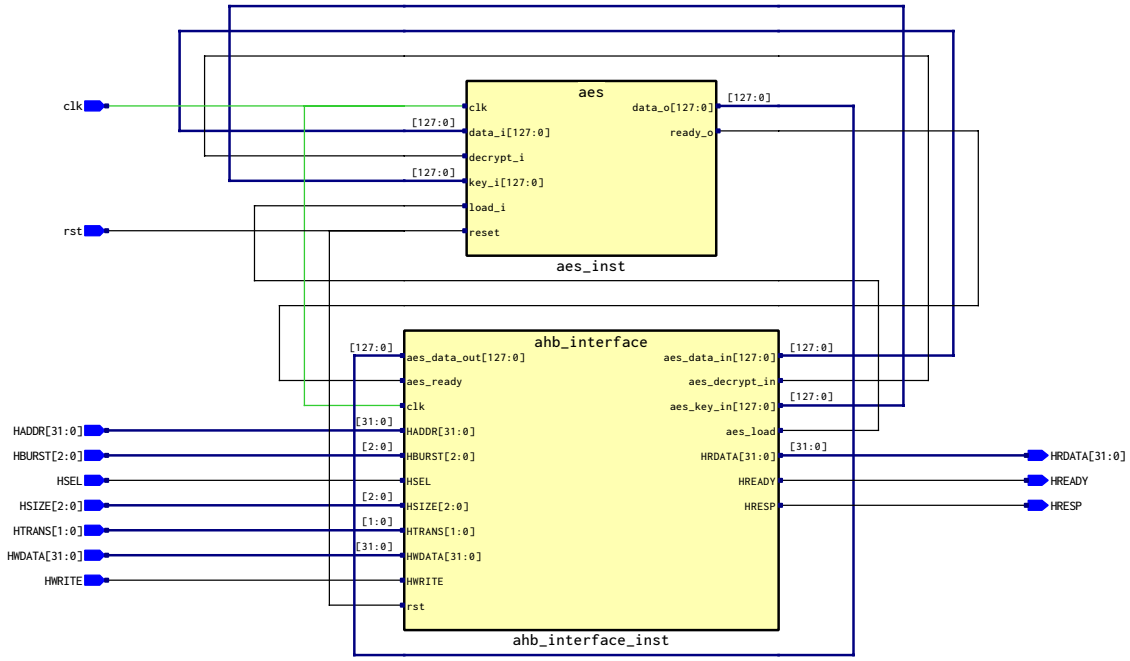


Figure 3.15: Top-level schematic view of AES + AHB-Lite

SIMON + AHB-Lite

Building on the AHB-Lite interface for AES, only a couple simplifications needed to be made to make it ready for SIMON. Because the SIMON block size is only 64 bits, bursts of two 32-bit beats are can replace the original 4-beat bursts in $WRITETEXT$ and $READTEXT$. Next, no selection on encryption or decryption is made as only encryption is currently supported by the supplied SIMON IP core. The FSM diagram in Figure 3.14 can thus also be applied to SIMON, with the removal of decryption/encryption selection states. Both AES and SIMON being block ciphers, the effort for creating a SIMON interface from an AES one is minimal. The resulting RTL schematic is shown in Figure 3.16.

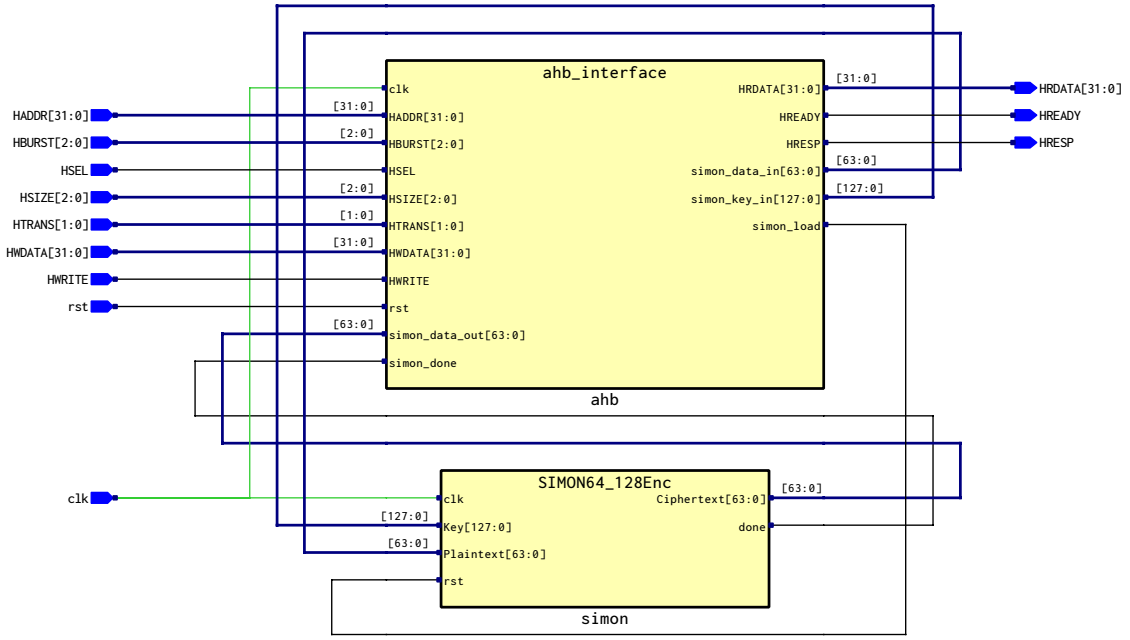


Figure 3.16: Top-level schematic view of SIMON + AHB-Lite

PHOTON + AHB-Lite
Because of the different nature of data input/output of the PHOTON IP core compared to AES/SIMON, more changes were needed to this AHB-Lite interface than going from AES to SIMON. The main problem is the 'streaming nibble' nature of data input/output. AHB-Lite data is clocked in at 32 bits per clock cycle, whereas PHOTON only takes 4 bits, a nibble, per clock cycle.

Next to the AHB-Lite interface itself, an extra interface block was introduced to take care of proper resetting, loading and reading data from the PHOTON IP core. This device integrates a small FSM of which a diagram is shown in Figure 3.17. Four states are present, of which $WRITE$ is the initial state. In this state, the initialization interface waits for its $init$ input bit to be asserted, indicating that the AHB-Lite interface has stable input data available that is ready to be hashed. Following init bit assertion, the FSM enters $WAITRESET$ in which a fixed number of cycles is spent waiting until the internal 36-nibble state matrix of the PHOTON IP core is fully initialized. A state transition to $SHIFTIN$ occurs after 38 cycles, determined empirically to be the lowest number of cycles in which the PHOTON IP core can be initialized. $SHIFTIN$ connects the nibble output of the AHB-Lite interface directly to the nibble input of the PHOTON IP core. During the time that the input nibbles are shifted in, the device remains transparent for writing but opaque for reading from the PHOTON IP core, as seen from the AHB-Lite module. During hashing, the device already enters the $SHIFTOUT$ state and waits until hashing is done by monitoring the $outReady$ signal coming from the PHOTON IP core. When hashing has finished, the device enables reading from but blocks writing to the PHOTON IP core and shifts out all 32 nibbles to the AHB-Lite interface.
Besides safe reading and writing of data, providing a simpler reset mechanism (one init bit) is also a task of this initialization interface. The proper sequence, together with the FSM states that are passed is shown in Figure 3.18. The signal $state\_q$ indicates in which state the device is. Compare this with the diagram of Figure 3.17. During state 1, the six columns of the PHOTON state matrix are initialized, as can be seen in Figure 3.18, the lower six wave forms. Asserting $nBlock$ and $nReset$ in state 2 signals the shifting in of data and in state 3, the given combination of $nBlock$, $nReset$ and $nInit$ indicates that hashing can and will take place.

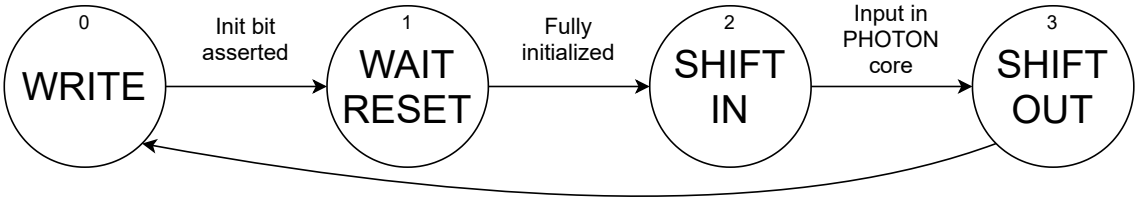


Figure 3.17: FSM describing the PHOTON initialization interface

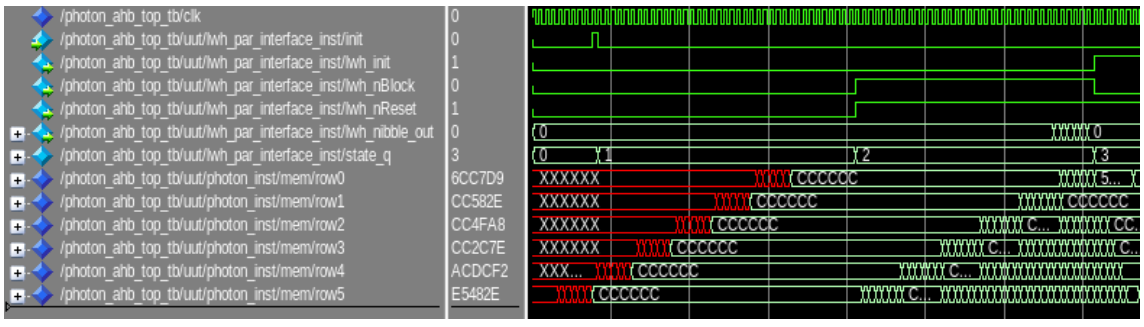


Figure 3.18: Top-level schematic view of AES + AHB-Lite

Adding AHB-Lite to the PHOTON IP core is substantially different than to AES or SIMON. Fortunately, with the addition of an external FSM as described in Section 3.3.2, only a single FSM has to be implemented in this particular AHB-Lite interface. As there is no encryption key present in hashing,

any key configuration state from AES/SIMON can be removed. Just like the previous two interfaces, the initial state of the FSM is a polling state, here called $SENSE$. It is not an idle state because shifting out nibbles to the initialization interface can happen here if $funnel\_shift$ is asserted.
Just like with AES/SIMON, all states can be entered from the initial state. To make optimal use of the high-bandwidth nature of AHB-Lite, it is not required to go back to $SENSE$ after exiting state 1 to 3. For example, $LOAD\_IV$ could immediately be followed by $READ\_STATUS$. After iterating on multiple designs, a shift register structure for the internal registers holding initial state matrix value (144-bit) and output state matrix containing the hash (144-bit) was chosen. Loading data with indexed part-select, where all register bits are directly connected to other logic was significantly more resource-hungry and hence discarded. In $LOAD\_IV$, the 144-bit input message register is written in a burst of four 32-bit and one 16-bit beats. Hence, all input data can be loaded within six clock cycles (adding the addressing phase of the first beat). $HASHOUT$ takes care of the inverse process: the 144-bit output register, which contains the resulting hash, is sent out the $HRDATA$ port in a burst of five consecutive beats. Note that the AHB-Lite FSM does not take care of reading or writing its 144-bit input and output registers, nibble by nibble. This is taken care of by the initialization interface in Section 3.3.2, which can manipulate the contents of these registers by the $funnel*$ signals. Although 128-bit values are used as input and output of the PHOTON IP core, 144-bit registers are used as four extra nibbles are present in the internal state matrix (row signals in Figure 3.18).
The total assembly of PHOTON IP core with its two interfaces is shown in Figure 3.20. Although the internals are greatly different comparing with AES and SIMON in Figure 3.15 and Figure 3.16, the top-level inputs and outputs are exactly the same, which is essential for protocol consistency and shows that the primary goal of interfaces is achieved: abstracting away any crypto IP core behind a standard protocol.

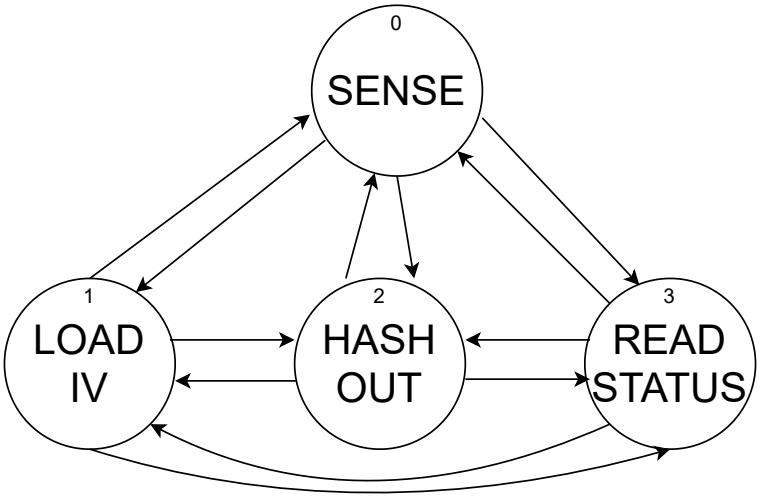


Figure 3.19: Simplified FSM describing the AHB-Lite interface for PHOTON

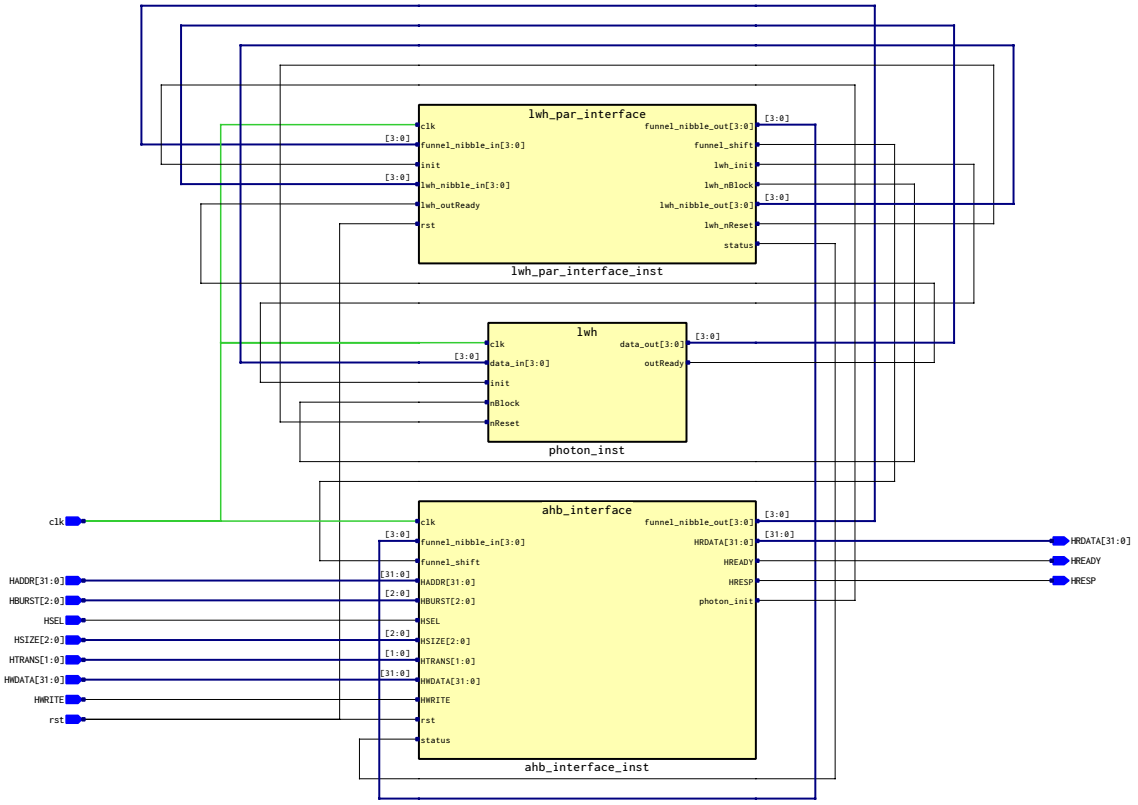


Figure 3.20: Top-level schematic view of PHOTON + AHB-Lite

3.4. Selected platforms

To gain insight in the feasibility of FPGAs in IMDs, our algorithms need hardware target platforms to be able to perform the required measurements. The roles of the *Extra Processing* block depicted in Figure 3.1 will be fulfilled by either an MCU (with and without hardware acceleration), FPGA or eFPGA platform running our designs presented in Section 3.2. For a baseline comparison, we select two MCUs that are very similar to the hardware of modern commercial IMDs. Next, an ultra-low-power (mW range) FPGA is chosen that will fit all hardware implementations of our algorithms. To be able to answer Question 2 listed in Section 1.2.1, three eFPGA fabrics were designed and included in the comparison. In the next subsections, the chosen hardware platforms will be discussed in detail and an introduction to the used eFPGA tools and architectures will be given.

3.4.1. MCUs

As reference platform on which software implementations of the aforementioned algorithms will be run, the EFM32 Tiny Gecko 11 is chosen [80]. This MCU has an ARM Cortex-M0+ at its core, paired with an AES-128 ASIC peripheral. With this device, performance with and without hardware acceleration can be benchmarked, in the case of AES-128. It will be interesting to see the impact of using hardware acceleration while also adding FPGA benchmarks to the equation.
Next, a higher performing MCU was selected to see how a step up in performance will cost in terms of energy consumption. The EFM32 Giant Gecko 11 [81] is in many ways very similar to the Tiny Gecko, with the main difference being an ARM Cortex-M4 which has replaced the Cortex-M0+. The same AES ASIC is present in the Giant Gecko as well. In Table 3.5 the most important properties of both MCUs are shown. A clock speed of 13MHz, the lowest available value, is set to approach the 12MHz used for the FPGA in Section 3.4.2.

Table 3.5: Specifications of selected Gecko MCUs

Name	CPU core	AES-128 ASIC	Op. Frequency (MHz)	Process technology (nm)
EFM32 Tiny Gecko 11	Cortex-M0+	Yes	12	TSMC 90nm
EFM32 Giant Gecko 11	Cortex-M4	Yes	12	TSMC 90nm

3.4.2. FPGAs

Choosing FPGAs for the comparison, the most important factor is high energy efficiency. As stated in Section 2.1.2, two families of FPGAs pose to be viable candidates for integration in IMDs looking at their physical size and power characteristics. The Microsemi IGLOO series has the advantage of being flash-based, leading to a low static power consumption. As static power consumption is the main contributing factor in overall energy consumption with low duty cycles (See Section 4.2.2), this would seem the logical choice over the other series, that of Lattice iCE40 UltraPlus FPGAs which is SRAM-based. Even better, both platforms could be taken and a comparison could be made between flash and SRAM-based FPGAs. However, due to the high costs of its hardware platform and tools licensing, the Microsemi IGLOO series is not included in our experiments. Instead, the more affordable iCE40 UltraPlus is set as the baseline FPGA platform, with approximations made later on of flash-based FPGA energy figures, see Section 4.2.2. Furthermore, the IGLOO series has no DSPs available, which are eagerly needed for artificial neural networks. One could do without them, like in [8], but at a high resource and energy cost. Added that the iCE40 UltraPlus has a well-documented CNN available and that tiny NNs are hard to get by, the Lattice platform is an overall better choice for our experiments. Table 3.6 lists the most relevant specifications of the selected FPGA. For development, Lattice's own software suite, Lattice Radiant, is used which incorporates all functionality expected from a modern FPGA design tool.

Table 3.6: Specifications of selected Lattice FPGA

FPGA	LEs (LUT4+DFF)	DSPs	RAMs	Process technology (nm)	Synthesis engine
iCE40UP5K	5820	8 (16x16)	4x 256Mb SRAM + 30x 4kb EBR	40nm	Lattice Synthesis Engine (LSE) Synopsys Synplify Pro

3.4.3. eFPGAs

Next to the off-the-shelf Lattice FPGA of Section 3.4.2, embedded FPGA technology delivered by Menta is selected for our experiments. For an extensive introduction to eFPGAs and the designed fabrics for our experiments, see Section 3.5.

3.4.4. LSE and Synplify Pro

Within Lattice Radiant, two synthesis tools can be selected. The default option is Lattice Synthesis Engine, which is developed in-house and is expected to perform especially well when calling on iCE40-specific hardware IPs. Next, Synplify Pro is available, which is an industry standard developed by Synopsys. All hardware designs created in this project are synthesized with both synthesis engines, to evaluate whether a particular engine should be used in certain cases.

3.5. Menta eFPGA

With eFPGAs, extra design steps are inserted before the regular synthesis and Place & Route toolflow. As the FPGA fabric is aimed at being embedded as an IP on the same die as other logic, it can be an arbitrary size. The eFPGAs in this comparison are not selected in the same sense as MCUs and eFPGAs which are off-the-shelf products, but designed to fit the selected algorithms. Before the designed eFPGA architectures are presented, an general overview of the design process will be given in Section 3.5.1. Next, the Menta eFPGA topology and architecture details will be laid out in Section 3.5.2.

Derivations that lead to the required resource amount for the designed eFPGA architectures are shown in Section 3.5.3. Finally, the designed and selected architectures are presented in Section 3.5.4.

3.5.1. Menta eFPGA design tools

Designing eFPGA architectures and actually implementing hardware RTL designs on them requires a special set of software tools. For our experiments, Menta made their eFPGA tools available. Their toolset consists of two programs: Origami Designer and Origami programmer. The names are self-descriptive: Designer is used to create eFPGA architectures, where Programmer bears more similarities with regular FPGA tools, featuring steps like synthesis, place & route and generation of bitstreams.

Origami Designer

With Origami Designer, eFPGA architectures of arbitrary size can be defined. A screenshot of the tool with an open project can be seen in Figure 3.21. With the menu directly right from the graphical eFPGA view, the size of the fabric can be determined as an X by Y grid. The number of clock/reset domains, I/O count and number of switchbox interconnect wires are also defined in the design process. The basic building block is a group of eight LUT6 cells tied to a single switchbox, but hard macros like RAMs and DSPs can be defined and placed in the fabric. The small yellow rectangles in Figure 3.21 indicate hard macros, placed column-wise in the first and last column of the defined fabric. One very important property of hard macro placement in Origami Designer is that the tool puts the maximum possible number of hard macros in a marked column, only leaving space for LUT6 blocks if further hard macro placement is impossible. More on this property and its design implications in Section 3.5.2.
Other features of Origami Designer include an estimation of the static power and area of the designed fabric with regard to the used manufacturing process technology model. When just an indication of the required fabric size for certain hardware IPs is needed, a suitable architecture can also be computed automatically based on one or more HDL applications. Basic synthesis and Place & Route functionality is also integrated, but for more advanced options in this field, Origami Programmer is used.

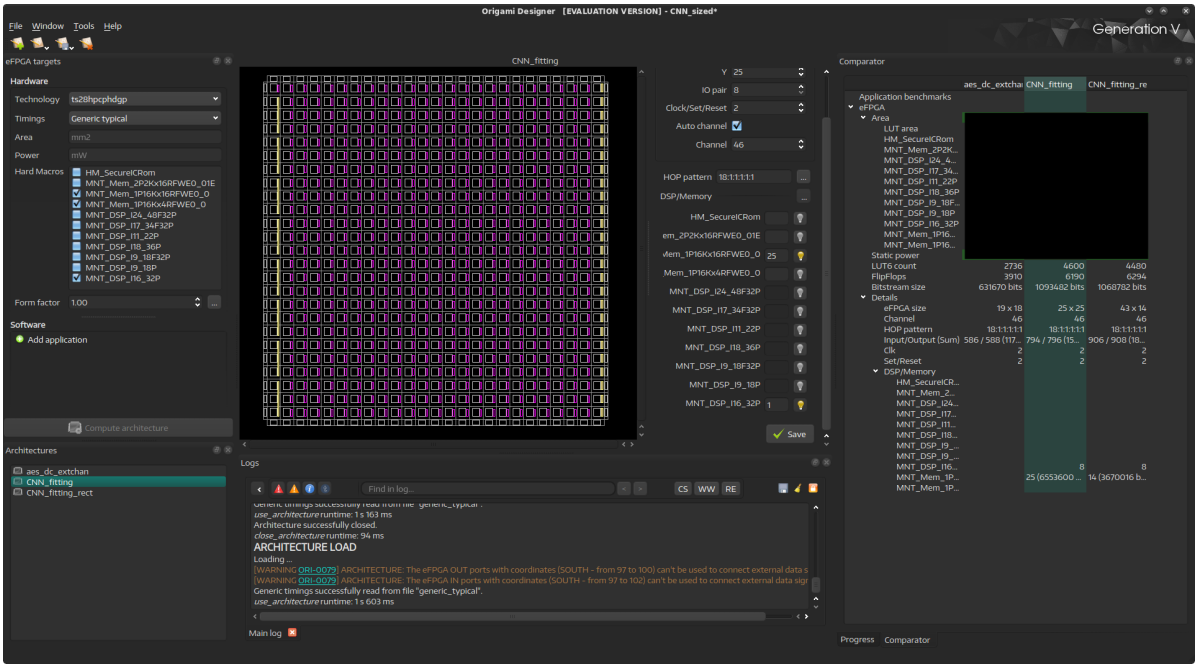


Figure 3.21: Origami Designer with an open project

Origami Programmer

Once one or more target eFPGA architectures have been created with Origami Designer, actual HDL applications can be implemented on those architectures with Origami Programmer. As has been mentioned before, this tool is very similar to comparable tools from FPGA vendors. The main difference lies in their target devices: with regular FPGA vendors, the target is one of their physical FPGAs, where

with Origami Designer the target architecture is a custom designed eFPGA. To run implemented HDL applications on the designed eFPGAs, the target device first needs to be manufactured after it can accept a bitstream. However, bitstreams can already be generated before manufacturing. In this thesis, we will only look at post-Place & Route results with regard to estimated static and dynamic power, physical area, timing and resource usage. Physical measurements on eFPGAs is considered out of scope, but those measurements will be done on the off-the-shelf Lattice FPGA selected in Section 3.4.2 and the experimental setup is explained in Section 3.6. In Figure 3.22, a screenshot of an open project in Origami Programmer is shown. Blue lines in the graphical eFPGA view are visualized interconnections between elements post-Place & Route, which give an indication of the relative resource occupation. Together with Origami Designer, an eFPGA architecture can be fit to a set of HDL applications without much effort, which was also done when designing our eFPGA architectures in Section 3.5.4.

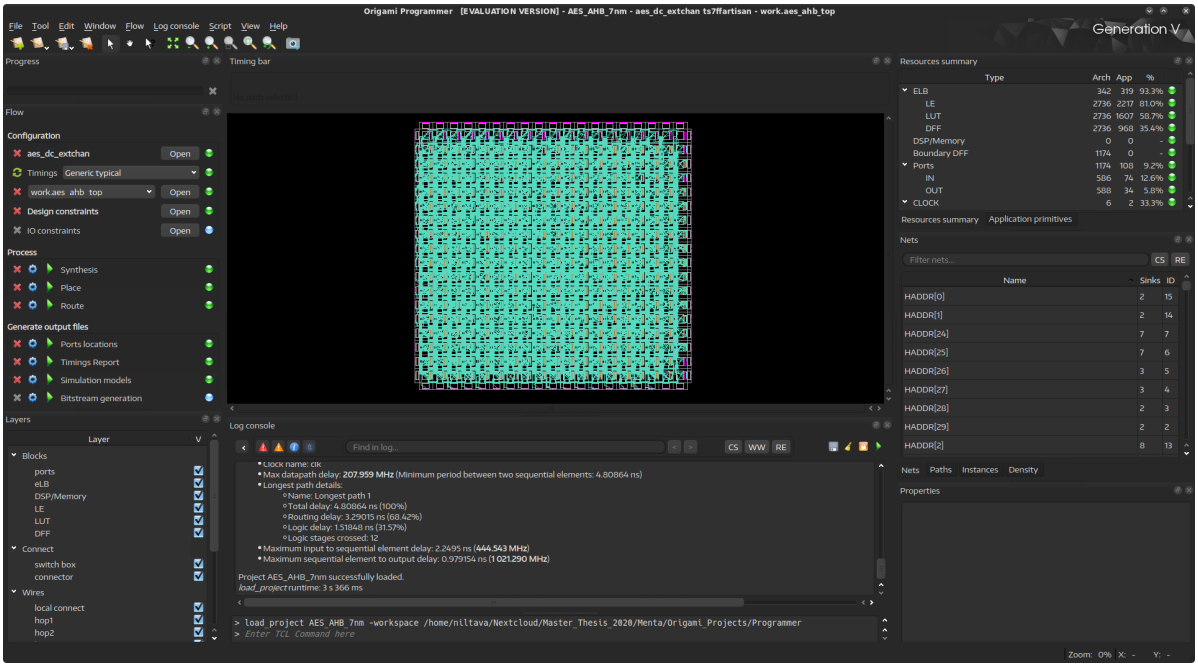


Figure 3.22: Origami Programmer with an open project

3.5.2. Menta eFPGA topology

With Origami Designer and Programmer, eFPGA architectures can be designed and hardware applications made fit on those designs. To get a deeper understanding of the different design parameters involved, this subsection outlines the most important eFPGA topology properties, as well as design characteristics that have to be taken into account.

Example Menta architecture

The simplest architecture, that is also the default configuration at a new Origami Designer project, consists of a grid of 3x3 Embedded Logic Blocks (ELBs), I/O pairs of width 8, a single clk/set/reset channel, interconnect channel width of 16 and no custom hard macros (like RAMs and DSPs). A graphical representation of this architecture is shown in Figure 3.23. As can be seen from the figure, an ELB consists of four MLUTs, which consist again of two LEs. MLUTs have two shared inputs for their two LEs, hence a grouping per two LEs is made. The channel width is equal to the number of interconnect wires (inputs and outputs) between SBs, added together for every of the four sides of an SB. For the example architecture, its channel width of 16 means 8 input and 8 output wires between all adjacent SBs. Every bit of I/O pair width corresponds to a buffered 1-bit input and output port.

Menta architecture with hard macros

Two architectures that would be fitting for the Lattice CNN were designed, of which one can be seen in Figure 3.24. More on the architectural design choices in Section 3.5.4. Next to ordinary ELBs

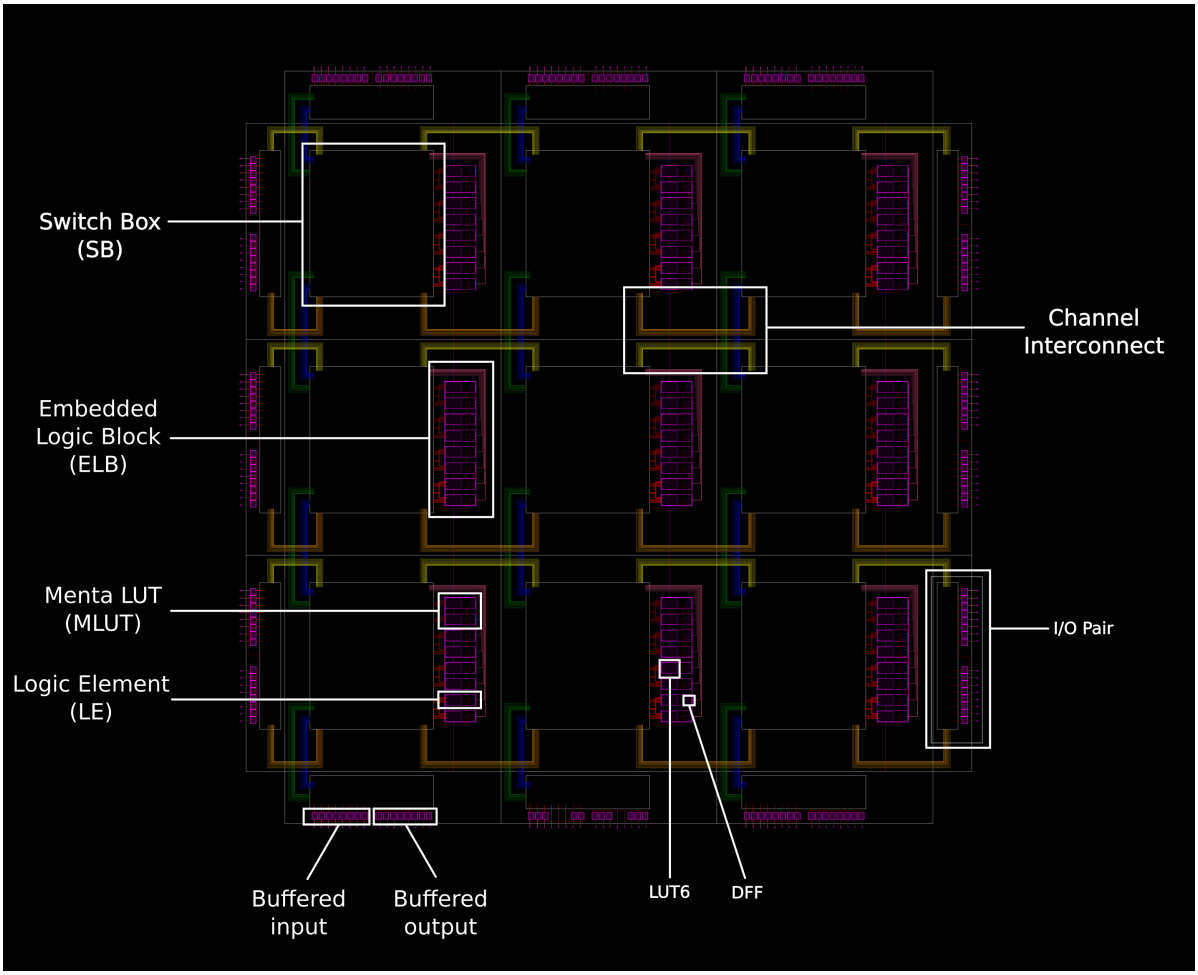


Figure 3.23: A reference Menta architecture, outlining its different building blocks

that contain LEs, switch boxes are assigned to connect to DSPs and RAMs. The number of SB that is occupied by a single DSP or RAM is determined by two factors: architecture channel width and DSP/RAM I/O width. Here, single port RAMs with 16-bit read-write ports only need one switch box, as the channel width of 46 provides for enough input (23) and output (23) wires to connect one RAM to one switch box. The DSP is a 16x16 one, meaning that it needs 32 input and output wires, 64 in total. This DSP configuration was selected to match that of the Lattice baseline FPGA. Origami Designer spreads it over three SBs, presumably so because the total number of wires in one direction is $23 * 3 = 69$ in this case, the smallest number of resulting wires that is larger than the total number of wires from the DSP.

Hard macro placement

Within Origami Designer, hard macros can be placed by marking columns to be available by one hard macro type per column. Different types of RAMs can exist in the same architecture, with different port widths, port numbers, capacities etc. but only one type of DSP can exist within an architecture. The number of DSPs and RAMs is not set directly but determined by how many DSP/RAM blocks fit in the marked columns. In the architecture of Figure 3.24, the first and last column are marked to be filled with 16x16 DSPs. As a DSP in this architecture needs three switch boxes, the maximum number of DSPs is automatically set to eight. The remaining switch boxes in the marked columns are set to regular ELBs, as is done here with the two top rows. As the RAM blocks used only occupy one SB and one column is marked, that whole column is filled with RAM blocks.

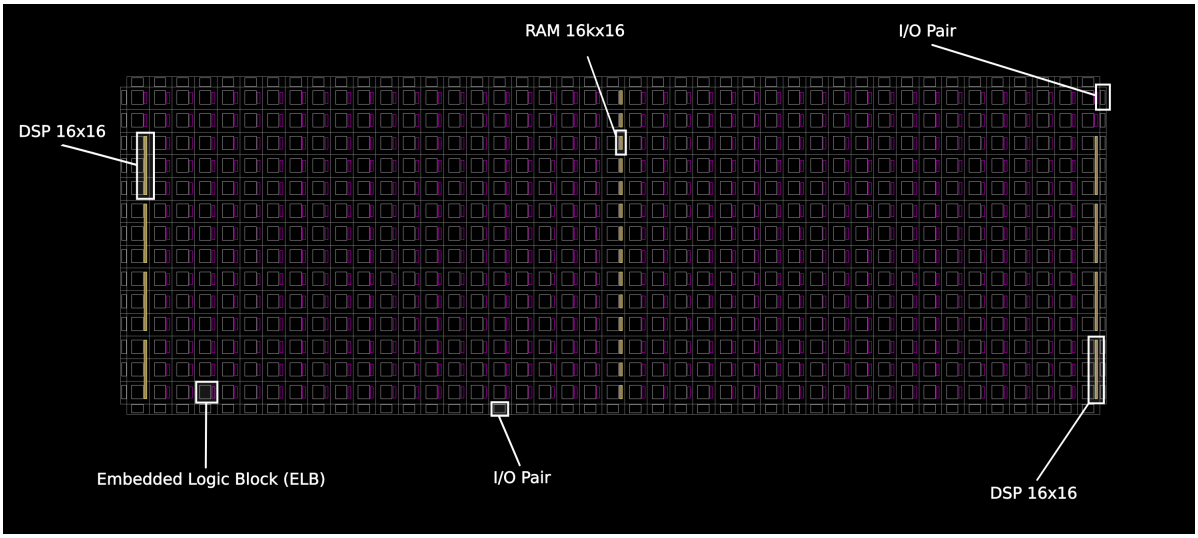


Figure 3.24: A CNN-fitting architecture with RAM and DSP hard macros

Logic Elements in DSP columns
Designing an eFPGA is done mainly by controlling its grid size as stated in Section 3.5.1. All inputs and resulting output parameters are listed in Table 3.7. After generating various architectures, further investigation uncovered a discrepancy between the reported number of LEs (and thus LUTs and DFFs) by Origami Designer and the actual number. The reason for this is that remaining LUTs, here listed as LEs_{NR}, are not counted towards the total resources. This can happen in the event that DSPs occupying more than one grid row are used and the remainder of Y mod SB_{DSP} is nonzero. The same can happen with RAMs, however no multi-column RAMs have been placed in our architectures. As an example, an architecture with four rows (Y = 4) and a column with a single DSP covering three rows will leave one SB in the DSP row available for an LE. It has been verified that these remaining LEs are used by the Place & Route engine. However, for simplicity, the official resource count as reported by the Origami tools is adhered to.

3.5.3. LUTs and DFFs calculation of Menta CNN numbers

As has been mentioned in Section 3.2.4, the Compact CNN by Lattice is proprietary and therefore only available on Lattice FPGAs. As a result, it is not possible to get actual resource usage numbers for the Key Phrase Detection project on Menta eFPGA fabric. To get an indication of the prospected resource usage and to size a suitable eFPGA architecture accordingly, a look is taken at the correlation between Lattice and Menta results for the security primitives, which can be Place & Routed on both platforms with success. Based on the calculated resource usage numbers in this section, the ”CNN-ready” eFPGA architectures are sized. The most important reason why there is no 1:1 resource usage pattern between Menta and Lattice, is because the first is a LUT4 and the latter a LUT6 architecture. Hence, a scaling factor for the number of used LUTs,going from LUT4 to LUT6, needs to be found. Menta already provides an equivalent #LUT4s with every architecture. The scaling factor used there is 1.52, meaning that a Menta architecture based on LUT4 cells would need 1.52 times as much LUTs as when using LUT6 cells. As we will see later, this number is a bit optimistic. For the complete resource usage analysis, see Section 4.2.

Resource usage ratios between Lattice and Menta
First, the ratios Menta/LSE and Menta/Synplify Pro are calculated for number of Logic Elements, LUTs and DFFs for all security primitive HDL implementations respectively. so this means six different blocks:

- AES-128 + AHB
- AES-128 + UART
- SIMON-64/128 + AHB
- SIMON-64/128 + UART

Table 3.7: Overview of Menta architecture design parameters

Variable	Description	Calculation
SB	Switch box, basic building block	X $*$ Y
X	# SB columns	$Input$
Y	# SB rows	$Input$
cols$_{DSP}$	# columns that are occupied with DSPs	$Input$
cols$_{RAM}$	# columns that are occupied with RAMs	$Input$
SB$_{DSP}$	# SBs occupied per DSP block	$Input*$
SB$_{RAM}$	# SBs occupied per RAM block	$Input*$
DSPs	# DSPs in architecture	$\lfloor \frac{Y}{SB_{DSP}} \rfloor$
RAMs	# RAMs in architecture	$\lfloor \frac{Y}{SB_{RAM}} \rfloor$
IOpair	# inputs & outputs per SB on architecture edge	$Input$
CLK$_{S/R}$	# clk/set/reset channels	$Input$
SB$_{HM}$	# SBs used for hard macros, either DSP or RAM	DSPs $*$ SB$_{DSP}$+RAMs $*$ SB$_{RAM}$
LUTs	# LUTs in architecture	$8 * ($X $*$ Y $-$ SB$_{HM})$
DFFs	# DFFs in architecture	LUTs $+4*$X$*$Y$*$IOpair$-5*$CLK$_{S/R}$
LEs$_{NR}$	# LEs (LUT+DFF) in HM column not reported	$8 * [Y($mod SB$_{DSP}) * cols_{DSP} + Y($mod SB$_{RAM}) * cols_{RAM}]$
LUTs$_{REP}$	# LUTs reported by Origami	LUTs - LEs$_{NR}$
DFFs$_{REP}$	# DFFs reported by Origami	DFFs - LEs$_{NR}$

- PHOTON-128 + AHB
- PHOTON-128 + UART

This results in six vectors with ratios between zero and one: two for LEs, LUTs and DFFs respectively, because calculations have been done separately for Synplify Pro and LSE results. Next, the median of all six vectors is taken per vector, leaving two medians per LE, LUT, DFF metric. This is to remove outliers, like the considerably lower number of DFFs for PHOTON-128_UART when switching the synthesis engine from LSE to Synplify Pro. The reason for this outlier is that Synplify Pro infers dedicated RAM blocks for this specific implementation where LSE can be set to map all memory definitions in registers. Every median pair (LSE and Synplify Pro) is averaged, leaving one median per metric (LEs, LUTs, DFFs). We will call each of these three medians 'mean medians' later on. To calculate the final number of LUTs, any obtained mean median is multiplied by the mean number of its respective resources between Synplify Pro and LSE, as found for the Key Phrase project. This is done for the number of LEs, LUTs and DFFs respectively.
The calculated number of resources for the Key Phrase project is less on the Menta platform than on Lattice as shown in Section 4.1.1. Note especially the number of LUTs, which is significantly lower. This is within expectation, as Lattice uses 4-input LUTs whereas Menta uses 6-input LUTs. From this calculation, it can be concluded that an eFPGA from Menta that has to fit the Key Phrase project only has to have around 4500 LUT6 Logic Elements, less than the 5820 of the iCE40UP5K FPGA.

3.5.4. Resulting eFPGA architectures

Since eFPGA architectures can be instantiated and modified with little effort, multiple architectures were created for our experiments. Due to the different resource usage nature of the security primitive IP cores compared to the Lattice Compact CNN, we opted for an architecture with and without hard macros. Specifications of all three architectures can be found in Tables 3.8 and 3.9.

Table 3.8: Input design parameters for the three Menta architectures designed for our experiments

Arch	X	Y	IOpair	$CLK_{S/R}$	$cols_{DSP}$	$cols_{RAM}$
1)	19	18	8	2	0	0
2)	25	25	8	2	1	1
3)	43	14	8	2	2	1

Table 3.9: Resulting properties of the three eFPGA architectures designed for our experiments

Arch	SB_{DSP}	SB_{RAM}	DSPs	RAMs	SB_{HM}	LUTs	DFFs	LEs_{NR}	$LUTs_{REP}$	$DFFs_{REP}$
1)	1	1	0	0	0	2736	3910	0	2736	3910
2)	3	1	8	25	49	4608	6198	8	4600	6190
3)	3	1	8	14	38	4512	6326	32	4480	6294

Architecture 1: cryptographic primitives
The first architecture was sized just large enough to fit AES + AHB and AES + UART. As will be shown in Section 4.1.1, either interface can be attached to the AES IP core without requiring a differently sized eFPGA fabric. No DSPs or RAMs are available in this architecture, as all security primitives are heavy on logic operations and do not make use of those special blocks when synthesized with Lattice Radiant. When AES-128 can fit, SIMON and PHOTON will fit just as well, as the latter two are designed to use significantly less resources.

Architecture 2: ready for neural networks
To potentially fit the Lattice Compact CNN, Architecture 1 is too small with 2736 ELBs. Calculations from Section 3.5.3 have determined that the required ballpark number of Logic Elements needs to be 4581. Architecture 1 was expanded to accommodate for this increase in ELBs, having 4608 in total. Additionally, eight 16x16 DSPs and 25 256Mb single-port RAMs were added, to have for both hard macro types at least the same number present in the iCE40UP5K FPGA. Note that the number of DSPs is identical, but the RAM amount is substantially higher. With a similar square shape (25x25) as can be noted from Table 3.8, marking a column for RAMs that occupy one SB will always result in 25 RAM blocks. Smaller RAMs could always be inserted to equalize the Lattice-Menta difference in this point. With a large number of RAM and DSP options, Menta provides power models to a customer when an eFPGA is actually going to be implemented and manufactured. As no implementation was planned for this thesis, no power model was made available. The Menta dynamic power estimations in Section 4.3.2 include only the projected LUT/DFF dynamic power.

Architecture 3: hard macro count reduced
Assuming that only 256Mb single-port RAM blocks are available, the number of them present in Architecture 2 needs to be reduced to bear closer resemblance to the iCE40UP5K FPGA. Because only one column is marked for RAM insertion, it is not possible to reduce the amount of RAMs simply by marking less columns. By resizing the fabric from a 25x25 square to a 43x14 rectangle, the column size is reduced to 14, implicating that only 14 RAMs are present. Reducing the fabric to four rows would result in an equalized amount of RAMs (4). However, this would make routing prohibitively difficult. The resulting size of 43x14 matches the number of ELBs of Architecture 2 the most, while simultaneously having a smaller number of RAMs for a more 'fair' comparison to the iCE40UP5K.

3.6. Current measurements

With our selection of algorithms, interfaces and platforms, current measurements are conducted to gather further insight in energy feasibility of typical use cases of reconfigurable fabric in IMDs. Measurements were performed on all physical platforms, meaning that both MCUs and the iCE40 FPGA were to run all three crypto IP cores. The Key Phrase Detector project, incorporating a convolutional neural network (CNN), only runs on the iCE40 FPGA as it is proprietary to Lattice. Therefore, no measurements could be done on the MCUs for this algorithm. Artificial neural networks that have an equivalent C++ and FPGA implementation were at the point of writing this thesis impossible to get

by. Developing or porting such a network was considered to be out of the scope of this thesis. As a result, no direct comparison of results from software MCU to FPGA implementations could be made. Current draw and energy consumption estimations were done for all eFPGA fabrics with tools supplied by Menta. An overview of the available measurement data points is shown in Table 3.10.
In the next subsections, the used measurement methods and necessary HDL modifications are laid out.

Table 3.10: Available measurement data points

Algorithm	MCUs	FPGA (iCE40)	Menta Arch 1-3
AES	yes	yes	no, estimated
SIMON	yes	yes	no, estimated
PHOTON	yes	yes	no, estimated
Key Phrase Detector	**no**	yes	no, estimated

3.6.1. MCU measurements

With fixed hardware in the case of MCUs, current draw depends mainly on the internal peripherals that are being used by a running program and the clock speed. Both Gecko platforms have an AES-128 ASIC, which greatly influences active current draw. In Table 3.11, all measured current draws for both MCU platforms are shown. All currents are MCU core currents only; the complete (development) board would have a higher current draw. However, having only core currents enables us to compare directly to the FPGA measurements. Static currents are based on the reported value in sleep mode $IEM4H\_VS$. When the dedicated AES ASIC is used, current draw increases but as will be shown later on in Section 4.2.2, energy consumption per encrypted block will be significantly less than a software run due to the drastically shorter execution time. Calculating consumed energy per encryption operation, the total execution time of one operation is counted and multiplied with the respective dynamic current value and the operating voltage, see Section 4.2.1. All dynamic current values are average currents over the duration of a single cryptographic operation, for all three crypto IP cores.

Table 3.11: Static & dynamic current values for Tiny & Giant Gecko @ 13MHz (in mA)

MCU	Static	Dynamic without AES	Dynamic with AES
Giant Gecko	0.021	1.44	1.98
Tiny Gecko	0.013	0.58	1.06

3.6.2. FPGA measurements

Obtaining dynamic current on the Lattice iCE40 platform is done by measuring the voltage across a 1Ω resistor specific test points on the used iCE40UP5K-B-EVN board, as marked in Figure 3.25. All applications were run continuously under full load during the measurements, giving an average dynamic current value in a similar way to 3.6.1. The measured current is of the whole FPGA package, but not the surrounding components like external flash. Multiplying with the duration of a single encryption operation and the operating voltage would yield the energy consumption for that specific design.

3.6.3. eFPGAs

Because the eFPGAs designed in this project do not exist physically, no direct current measurements could be done on these architectures. However, static power per architecture can be obtained from Origami Designer (Section 3.5.1) and for dynamic power estimations, Menta provided an experimental calculation model. With this model, dynamic power can be estimated with only three main input parameters: fabric size, clock speed and average activity factor. Process technology is also an important factor, which was fixed at 28nm TSMC technology. Note that no mention is made of a particular application that determines the power: this is simplified and replaced by the average activity factor. In this project, the relative resource occupation of each HDL implementation with regard to Architecture 1 is taken and used as a multiplier in the calculation. Otherwise, all applications would yield the same dynamic power, as the model normally expects the complete eFPGA fabric to contribute to the average activity factor.
As no real average activity factor could be extracted for any implementation in the duration of this

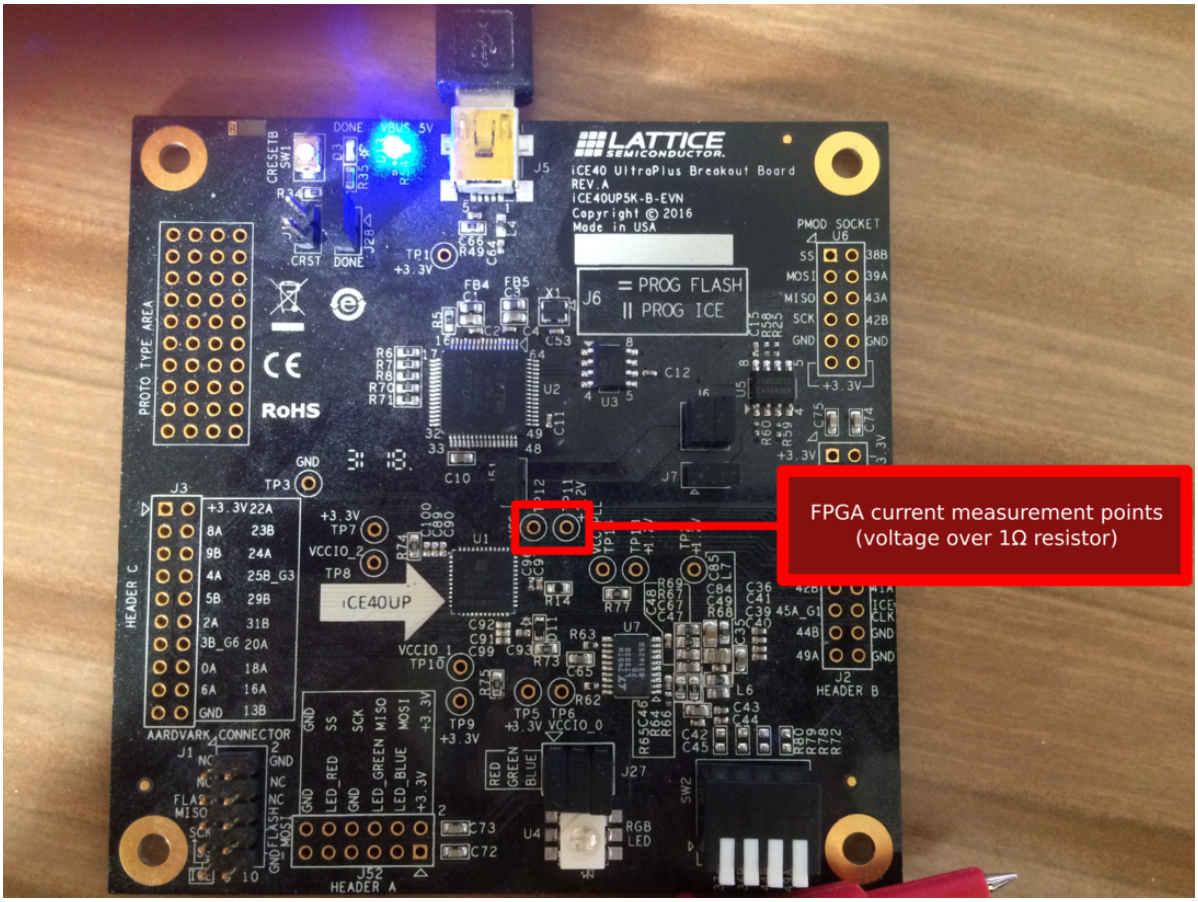


Figure 3.25: Current measurement points on the iCE40UP5K-B-EVN evaluation board

project, multiple realistic values are assumed in the calculations, see Section 4.3.2.

3.6.4. Crypto cores measurement preparations

To ensure reliable average dynamic current measurements, all crypto IP + interface assemblies needed slight modifications. If bitstreams would be generated of all crypto IP + interface cores described in Sections 3.2 and 3.3, only static power would be measured as the cores are not configured to do anything without specific inputs. First, the number of cycles was determined that each crypto IP core requires to encrypt a single block (128-bit for AES and 64-bit for SIMON) or hash a 128-bit input message (PHOTON). The resulting cycle count can be found in Table 3.12. For every block, counting was done from rising edge of its init signal to rising edge of its done signal. Note that only the crypto IP cores are taken into account here: communication delay from AHB-Lite or UART is not put in the equation, as the most influential factor on current draw would become the baud rate for UART and negligible for AHB-Lite.

Table 3.12: Cycle count for single cryptographic operation

Algorithm	Cycle count	Single operation definition
AES-128	505	Encrypting 128-bit plaintext
SIMON-64/128	45	Encrypting 64-bit plaintext
PHOTON-128	1073	Hashing 128-bit input message

With the cycle count for each crypto IP core from Table 3.12, PWM counters were inserted to trigger the load signal with the respective cycle count period. AES is taken as an example here and the same methodology is applied to SIMON and PHOTON. For AES, the PWM counter triggers the $init$ signal of the AES IP core every 505 cycles to ensure that the complete encryption cycle is executed and that it runs indefinitely, encrypting a hardcoded plaintext. Measuring is made easy this way: a single average current value is enough to derive energy consumption for a single encryption run because its number of cycles is known. See the modified RTL schematics for AES + UART and AES + AHB-Lite in Figure 3.26. In all implementations, a $stresstester$ block is inserted. This block is the PWM counter that triggers operation with the right period. Next, all implementations that have AHB need an extra trick in the form of the $ahb\_wire\_eliminator$. This block obscures the $HRDATA$, $HWDATA$ and $HADDR$ ports from the Map and Place & Route engine so that the design can be loaded on the Lattice FPGA. The iCE40UP5K only has 39 I/O pins, hence this modification is needed. As no communication takes place (input plaintext is hardcoded), this modification proves no impediments to our measurements.

3.6.5. Compact CNN accelerator measurement preparations

Because the Key Phrase detector project is already built to continuously monitor and listen to its inputs, no modification is needed to get it out of an idle state. The example bitstreams are loaded onto the iCE40 FPGA, after which average dynamic current measurements can be done in the same way as with the crypto IP cores in Section 3.6.4, while the design runs at 40 key phrase evaluations per second.

3.7. Conclusions

In this chapter, our benchmark suite consisting of four algorithms, two interfaces and three groups of platforms is presented and designed.

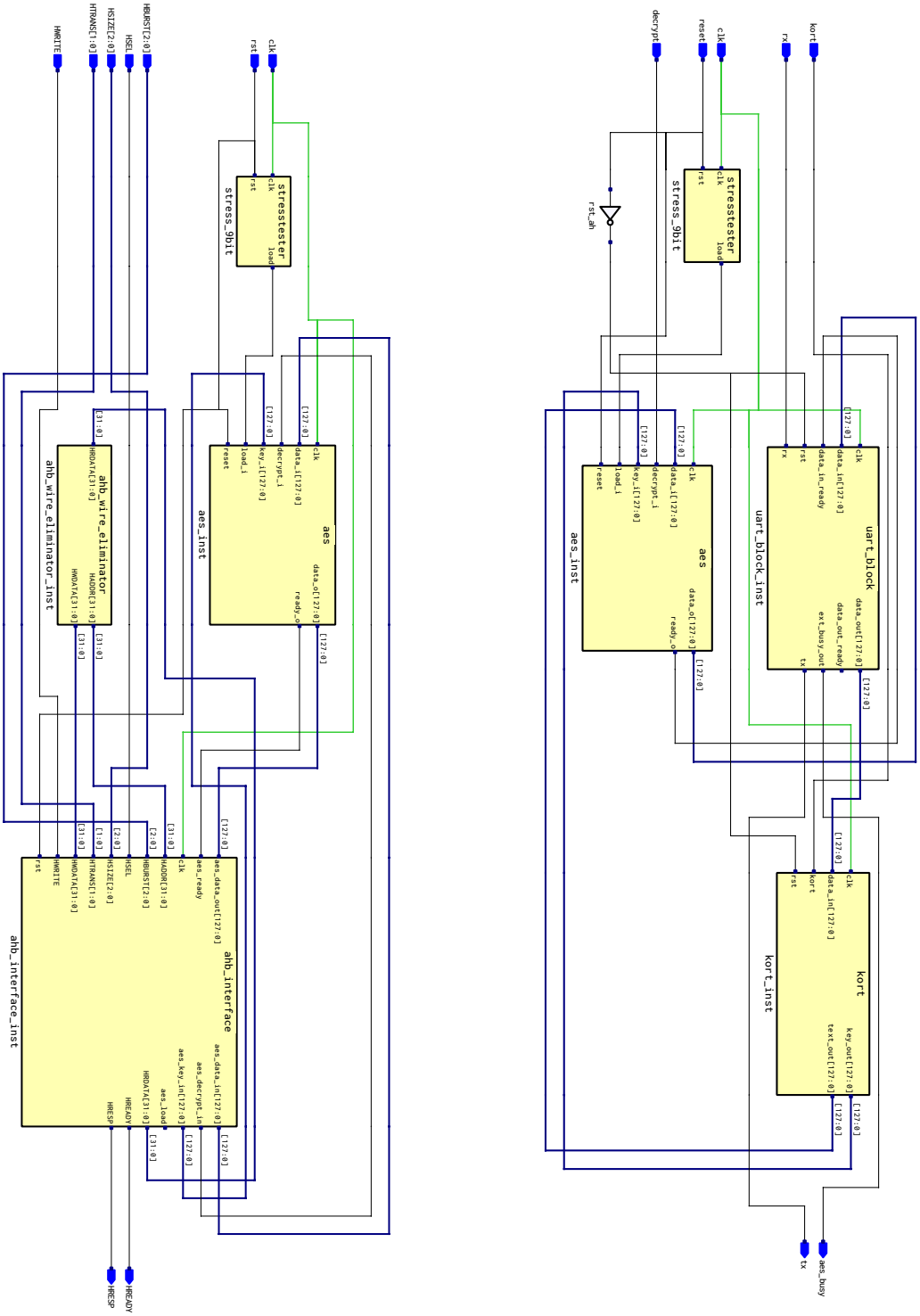
First, the implementations of the three selected security primitives (AES-128, SIMON-64/128 and PHOTON-128) are demonstrated. For every primitive, a premade IP core was selected that is optimized for low resource usage. With AES and SIMON being both block ciphers, interfacing them is done by presenting block and key data in parallel. Reading data from and writing to the PHOTON block is done in a serial fashion, as is fitting for hashing protocols which have a varying input message size. As neural network representative the Key Phrase Detector example project from Lattice is chosen, which is the last of our algorithm selection. The actual interfaces were created in Section 3.3, for simulating realistic resource overhead on the security primitives implementations. Two different scenarios were considered, as seen from the FPGA: off-die communication with an external host MCU and on-die communication in the case of an eFPGA integrated within an MCU on the same die. For both scenarios, fitting hardware implementations have been designed in the form of UART and AHB-lite interfaces. Where the UART interface is based on a simple 8-bit UART IP core, the AHB-Lite interface has been designed completely from scratch, with both interfaces requiring adaptations depending on the respective cryptographic IP core.

In Section 3.4, a set of two MCUs and a baseline FPGA have been selected to perform performance and energy comparisons for our experiments. Additionally, eFPGA technology will be considered. As is clear from the available ultra-low-power FPGAs currently on the market, the iCE40 UltraPlus series is the only affordable option. The MCU selection is based upon processors common in modern IMDs, with a higher end and energy-efficient version of the EFM32 Gecko series. This set of platforms is representable hardware for comparing a hypothetical FPGA-equipped IMD with a conventional MCU-based unit.

Next, eFPGAs are added to our platforms comparison in addition to MCU and FPGA hardware. An extra step is introduced in the design flow compared to regular FPGAs: that of designing the eFPGA fabric itself. With the Menta architecture being LUT6-based, the resource usage is different from our algorithms implemented on the baseline FPGA. Determining resource usage of the Key Phrase Detector on Menta fabric can only be calculated, as its HDL code is proprietary and non-portable. With an estimation based on linear trends between Lattice and Menta resource usage of all security primitives, the prospected LUT6 resource usage factor for the Key Phrase Detector is derived as being 75.33% of the LUT4 count on the Lattice FPGA. Having obtained this factor, eFPGA architectures that can just fit this CNN project were designed, as well as one fitting all security primitives, totaling three Menta architectures to be used in our experiments.

Last, our benchmark suite for current measurements on the iCE40UP5K-B-EVN platform is prepared.

This involves modifying our security primitive HDL code to let them run at continuous load, that is, encryption. For the Key Phrase Detector project, no modifications were necessary as this design is already built to operate continuously.
MCU current measurements are to be performed by taking active current, which is identical for all implementations as its hardware is unchanged. Only measurements of the security primitives can be done, as no Key Phrase Detector software version is available.
Physical eFPGA measurements are not possible as the architectures do not exist physically. Complete physical properties will only be available after hardening the eFPGA IP, which is not planned for this thesis. However, static power can be obtained from Origami Designer and dynamic power from an experimental calculation model. As a global average activity factor needs to be provided in this model, multiple realistic values are assumed.



(a) AES + AHB

(b) AES + UART

Figure 3.26: Modified RTL schematics for AES current measurements

4

Results

In this chapter, the research questions posed in Section 1.2.1 are revisited and answered through results of our experiments, for which the benchmark suite was prepared in Chapter 3. The goal of all experiments is to arrive at a clear picture of how and in what ways FPGAs could be a feasible and advantageous addition to IMDs. Therefore, typical use cases were implemented on FPGA fabric: AES, SIMON and PHOTON security primitives paired with UART and AHB-Lite interfaces, and the Lattice Key Phrase Detector project with a CNN inside on the iCE40UP5K FPGA. All three security primitives were run continuously on the iCE40UP5K-B-EVN development board as listed in Table 3.12. First, in Section 4.1 a look is taken at FPGA resource usage of each implementation to determine the needed eFPGA architecture sizes and to simplify and reduce the number of data points. Next, execution time and energy are evaluated in Section 4.2. Reduction in execution time is one of the main advantages to deploy an FPGA instead of a software implementation. This performance gain will come at the price of a higher energy consumption. To see whether extra energy consumption is tolerable, projected battery life is calculated from execution time and dynamic current/power draw. With battery life figures, it is easy to judge FPGA feasibility. Next, we take a look at eFPGAs. When area is of utmost concern, eFPGAs can be a good alternative to classical FPGAs because of integration within an MCU, hence the name embedded FPGAs. Added to that, energy consumption differences with the Lattice FPGA are investigated to see how much energy is saved by going from FPGA to eFPGA. Power and energy calculations are done with help of Menta-provided calculation tools as described in Section 3.6.3. Advantages in area and power consumption will be laid out in Section 4.3.
To answer if reconfiguration is feasible, we have to look at energy consumption per FPGA reconfiguration. No partial reconfiguration is possible, as that is currently reserved for more high-end and thus bigger FPGAs. Section 4.4 will show the tenets of IMD-FPGA reconfiguration.
Finally, a non-technical but highly important aspect is that of legal certification. No technical experiments have been done to answer the fourth question, but sources will show in Section 4.5 that no hindrance is to be expected in this field for FPGA-equipped IMDs.

4.1. Calculating and Reducing Results

4.1.1. FPGA resource usage

As noted in Section 3.5.3, an analysis of resource usage on the Lattice and Menta platforms is needed to determine the eFPGA fabric size for a neural network implementation. A first look is taken at the resource usage of all designs presented in Chapter 3. A design should not exceed the maximum available resources of the iCE40UP5K FPGA used in our experiments. Resource usage has a direct impact on energy usage: the more resources are active at a given time, the more energy is consumed. Added to that, exceeding the available resource limit of a low-powered FPGA forces the designer to opt for a higher end and more energy-wasteful FPGA. For this reason, a section is devoted to resource usage. We will see that many variables, such as synthesis engines and interfaces can be eliminated due to negligible resource usage differences. LUT and DFF resource usages can be found in Figure 4.2 and hard macro usages (RAMs and DSPs) in Figure 4.3. As can be seen from Figure 4.2, AES-128 is the largest crypto core as expected with 2358 to 2716 Logic Elements, depending on the interface and

synthesis engine used. SIMON and PHOTON occupy significantly less Logic Elements, with 668 to 776 and 578 to 672 respectively. This means that going from the de-facto industry standard in block ciphers (AES) to a lightweight block cipher (SIMON) results in a factor 3.5 decrease in resources! PHOTON is similarly sized to SIMON, indicating that lightweight hashing is just as feasible as a block cipher on small FPGAs.

LSE or Synplify Pro?
If we look at the differences between the two synthesis engines, LSE gives the best results with the Key Phrase detector, while Synplify pro consequently outperforms LSE where crypto cores are considered. This leads to the conclusion that LSE works best with complex Lattice-proprietary designs, which seems reasonable considering that LSE is an in-house synthesis tool. On the other hand, Synplify Pro is better with simpler, pure logic designs like all crypto cores plus interfaces. Nevertheless, the difference between synthesis engines is within 15% if the number of LUTs is considered, which is not that substantial. Therefore, in further results analysis no distinction will be made between Synplify Pro and LSE. As the Key Phrase Detector only gave valid current measurements when synthesized with LSE, the numbers of Synplify Pro were dropped. All further mentions of designs and measurements on the iCE40UP5K FPGA are thus LSE-synthesized implementations.

Interfaces
Looking at interfaces, UART and AHB-Lite occupy a similar amount of resources, with even smaller differences than choosing between LSE or Synplify Pro! AES having the greatest difference between interfaces with PHOTON being virtually the same. Therefore, like with different synthesis engines, no distinction between interfaces will be made in the following results analysis. Neither one will be dropped, but the average per crypto core will be taken, so AES-128 in following graphs is the mean between AES-128 + UART and AES-128 + AHB and the same goes for SIMON and PHOTON. Another important conclusion to draw from this observation is that an eFPGA, specifically designed to fit crypto cores, is not dependent in size on which interface is used. As a protocol like UART is more suited for communication with external devices, as seen from the host processor, it is more likely to be used for external FPGAs. Likewise, eFPGAs will be on-die and connected with a protocol like AHB-Lite. Therefore, no distinction in resource availability has to be made between eFPGAs and FPGAs, as their corresponding interfaces take up about the same amount of resources.

Hard macros
Where AES, SIMON and PHOTON do not occupy any hard macros, the Key Phrase Detector project makes eager use of them. Therefore, an FPGA architecture implementing a CNN should have plenty of RAMs and DSPs available. In Figure 4.3, the difference between SRAM and EBR configurations of the internal CNN is shown. Selecting slower (SRAM) or faster (EBR) RAM has no significant impact on Logic Element usage and maximum operating frequency hovers between 32MHz and slightly over 34MHz as seen from Figure 4.1. Both operating frequencies are royally above the required 12MHz and less than 5% apart, giving good reason again for a simplification of results. All following results will use the SRAM version of the design and Menta architecture 2 and 3 (see Section 3.5.4) are designed to have the same SRAM blocks that are used by this implementation present.

Menta Key Phrase Detector resource usage
As has been mentioned in Section 3.5.3, resource usage figures for the Key Phrase Detector project on Menta eFPGA architectures are estimated by calculation. As the iCE40 architecture is based on LUT4 blocks and Menta on LUT6, it cannot be said that the number of LUTs used by a certain implementation on the Lattice FPGA will equal that of any Menta eFPGA. We repeat the procedure of Section 3.5.3 here, but with real result figures now. In Table 4.1 the relative number of LUT6s occupied on Menta fabric compared to LUT4s on the iCE40UP5K FPGA is shown. Only calculations on LUTs are shown here, but the procedure is identical for DFFs and LEs. If we take the first entry, AES-128 + AHB, implementing that core on Menta fabric yields a LUT count of only 60.94% of that of the same core on the iCE40UP5K FPGA. Reason for this significant reduction is because Menta fabric is LUT6 based, where Lattice fabric consists of LUT4s. Taking the median of all implementations and the mean between the two synthesis engine options, we can conclude the LUT count of a crypto core will be about 75% on Menta fabric if we take 100% as the Lattice LUT count. This calculation is applied for LEs, LUTs and DFFs separately

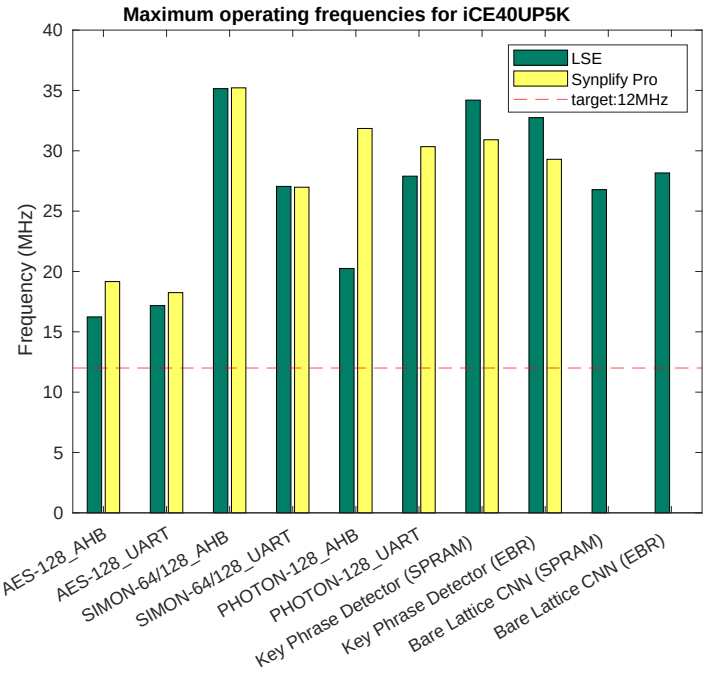


Figure 4.1: Maximum operating frequencies of all crypto core assemblies on the iCE40UP5K FPGA

Table 4.1: Obtaining the estimation factor for resource usage of CNN on Menta fabric

Implementation	LUT occupation on Menta, compared to: LSE	Synplify Pro
AES-128 + AHB	60,94%	70,05%
AES-128 + UART	72,76%	74,23%
SIMON-64/128 + AHB	86,06%	94,83%
SIMON-64/128 + UART	68,99%	72,29%
PHOTON-128 + AHB	91,68%	106,13%
PHOTON-128 + UART	75,15%	79,19%
Median (column-wise)	73,96%	76,71%
Estimation factor (mean of medians)	**75.33%**	

and results in the Menta Origami bars for the Key Phrase Detector Project in Figure 4.2. Interesting to note is that only the LUT count differs significantly from the Lattice platform, LEs and DFFs are virtually equal. For DFFs this makes sense, as no changes are made to flip flops when going from a LUT4 to a LUT6 architecture. For LEs, which consist of one LUT + one DFF, one would expect also a reduction in line with LUTs, as the number of DFFs does not dominate the number of LEs. This must be due to a different nature of Menta and Lattice synthesis engines and architectural differences.
For the hard macros, no calculations were done but the same number of them was assumed to be used on FPGA and eFPGA platforms, as the RAMs and DSPs in Menta Architecture 2 and 3 are made to be identical to those present in the Lattice architecture. Seeing that the Key Phrase Detector uses all eight DSPs on the target platform, it could be suspected that the algorithm is limited by its architecture. Using more than eight DSPs in a Menta architecture might bring the used number of LUTs further down from 75%, posing an even more advantageous situation for the eFPGA implementations. However, this situation will not be elaborated on in this thesis as it cannot be supported with quantitative statements.

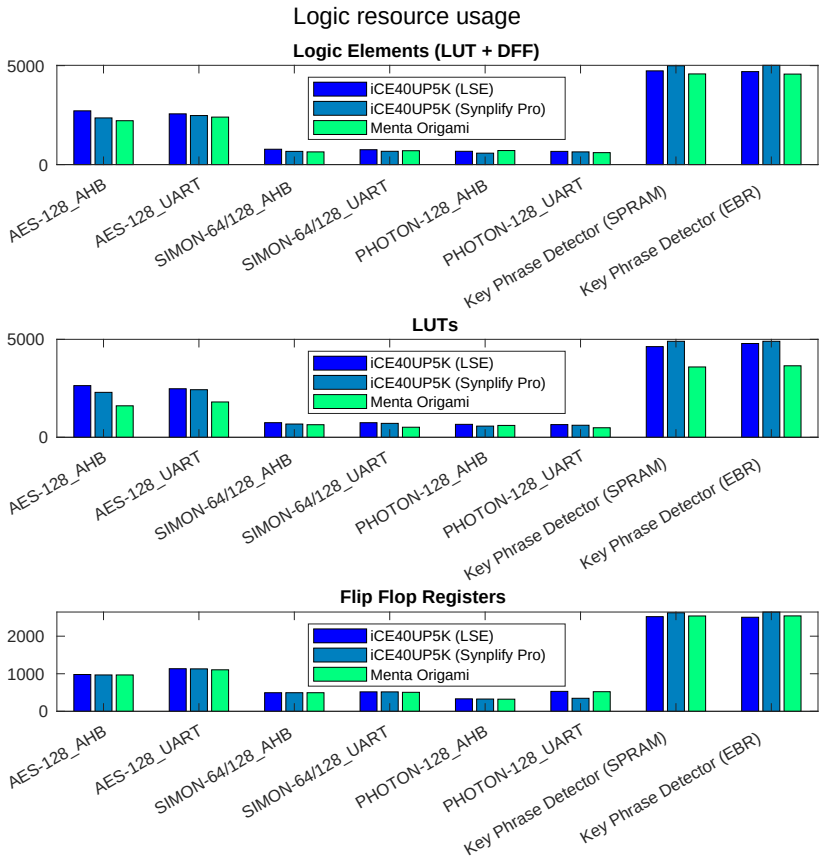


Figure 4.2: LUT and DFF resource usage on iCE40 FPGA and Menta eFPGA platforms

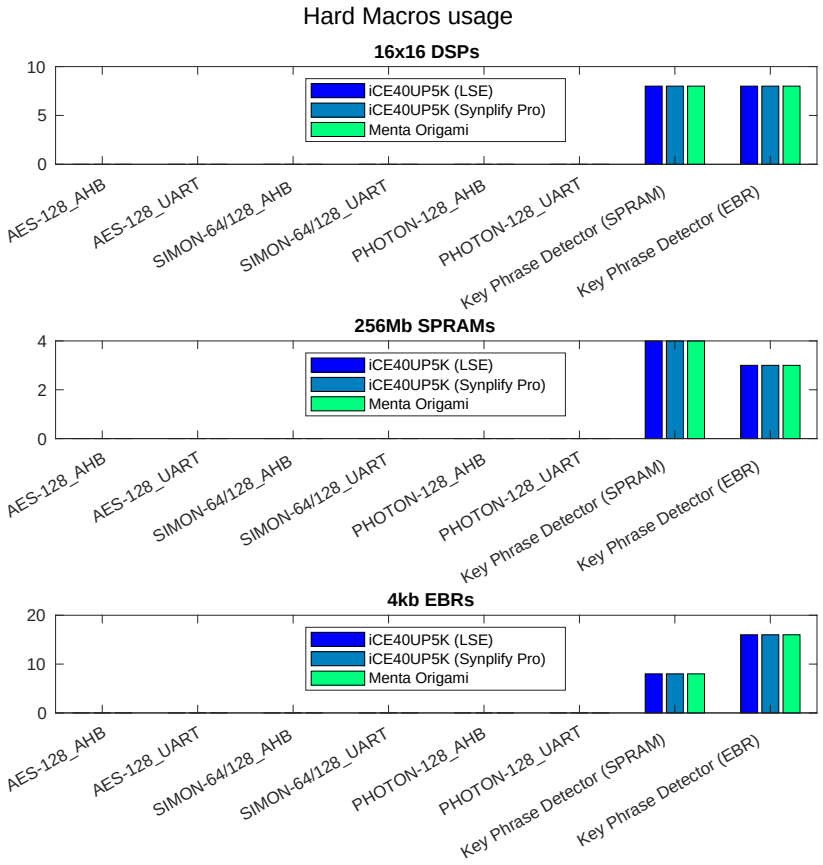


Figure 4.3: DSP and RAM resource usage on iCE40 FPGA and Menta eFPGA platforms

4.2. Q1: Can FPGA fabric be used in IMDs in terms of energy and execution time?

4.2.1. Algorithm execution time

While resource usage metrics give a qualitative answer to the feasibility of FPGAs in IMDs, execution time of the various algorithms in our experiments will show the significant advantage of reconfigurable fabric in this realm. In Figure 4.4 the execution time of a single operation, as defined in Table 3.12, is shown. The used clock frequency is 12MHz for all platforms. Three crypto algorithms were used, but four graphs can be seen: AES-128 can run in software (SW) and hardware (HW) mode on both Gecko MCUs, in the latter case using a dedicated CRYPTO peripheral. In all graphs, FPGA execution times are based on the number of cycles defined in Table 3.12 and a clock speed of 12MHz.
In the three SW graphs, it can be seen that the FPGA implementations vastly outperform all MCU software implementations. This is to be expected but gives way to the conclusion that fast cryptography needs customized fabric. Comparing the AES ASIC to the FPGA, again expected results: the FPGA is 17% to 40% slower than the ASIC. But this is not the whole story: from our algorithms selection, the ASIC integrated in the Gecko MCUs only supports a select range of primitives of which AES-128 is one, where the FPGA will also run SIMON, PHOTON and numerous other future crypto cores. Therefore, a performance penalty of up to 40% extra execution time has to be taken if reconfigurability is preferred over a rigid design. On the other hand, having flexibility like software implementations is achieved with FPGAs while vastly outperforming said software implementations. Based on these execution times, FPGAs would be viable candidates to deliver considerable performance gains without losing the possibility to upgrade or replace functionality in the future.

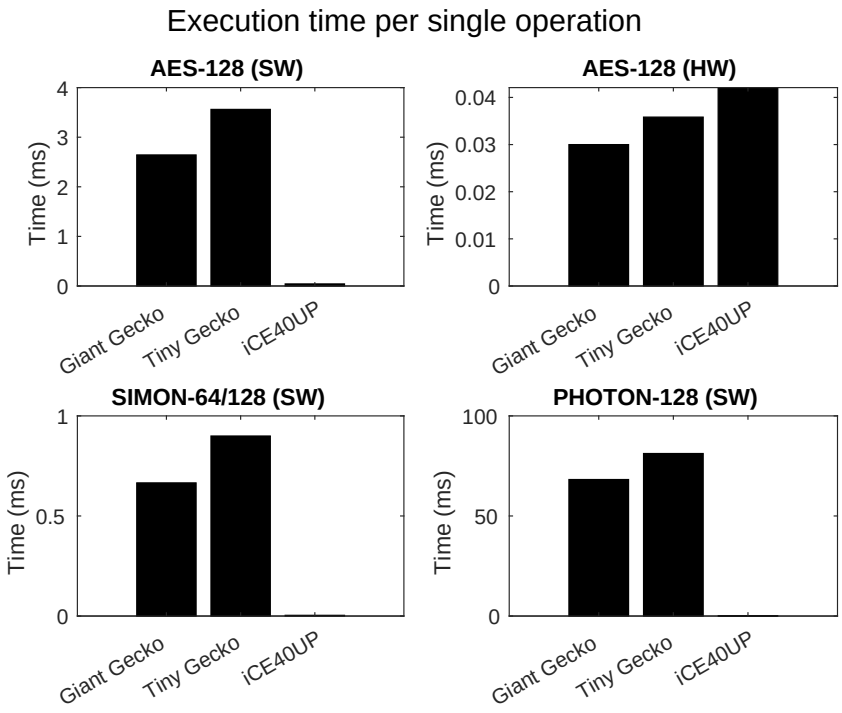


Figure 4.4: Crypto IP core execution time on MCU and FPGA platforms

4.2.2. Energy consumption

Considering the generally limited battery capacity of IMDs, even more important than execution time is energy consumption. A gain in performance does not necessarily result in a more energy efficient design, quite the contrary most of the time. In the next sections, a look is taken at energy consumption of all four algorithms used in our experiments to see if the advantages of FPGAs outweigh the extra energy consumption in an IMD environment.

Security primitives

First, the three security primitives are compared across the MCU and FPGA platforms. In Figure 4.5 the same setup is shown as with the execution times of Figure 4.4, but energy consumption is now graphed. With regard to the comparison of SW and FPGA runs, it can be said that the FPGA is not only orders of magnitude faster but also consumes significantly less energy for the same functionality. Especially PHOTON is an energy hog on the Geckos, consuming about a hundred times more energy than SIMON! Considering MCU hardware acceleration of AES-128, the FPGA consumes 4.4 times as much energy as the Tiny Gecko. From these observations it can be concluded that an FPGA will be more energy efficient and faster than SW by a large measure. Therefore it will be feasible to add an FPGA if no hardware acceleration is already available. But, an ASIC will not only be faster but also significantly more energy efficient. Having the luxury of reconfigurability using an FPGA will cost 4.4 times the energy of a dedicated ASIC when block ciphers are considered. With hashing, adding an FPGA becomes even more attractive as the energy gap between software MCUs and FPGAs widens.

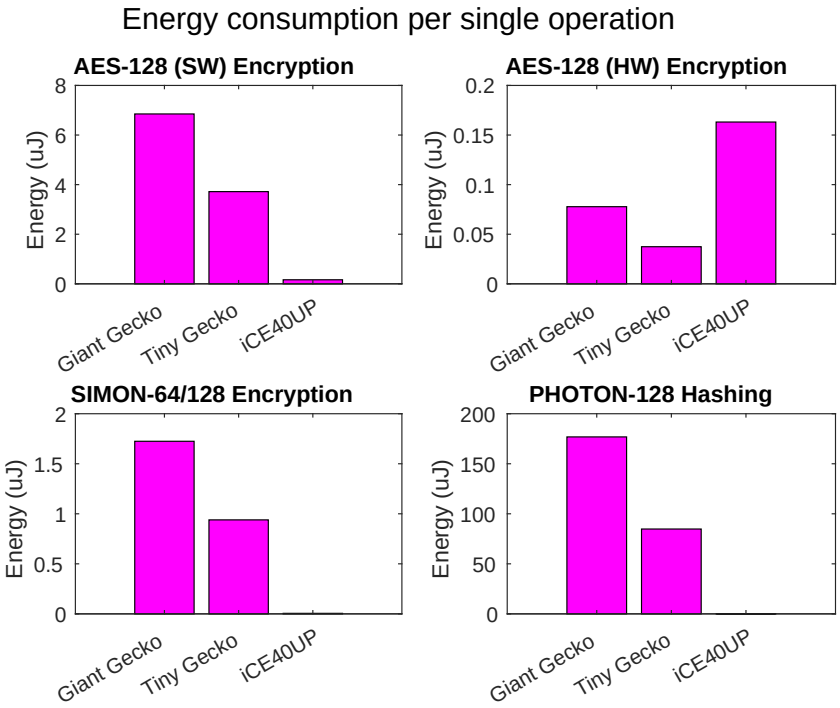


Figure 4.5: Crypto IP core energy consumption, MCUs and FPGA

Lattice CNN

As the Key Phrase Project is designed to be an always-on system, continuously listening for key phrases, no 'single operation' like for the crypto cores can be defined here. However, to be able to compare energy consumption of the FPGA implementations of all four algorithms, the runtime of the Key Phrase Detector was set to equal that of AES. In Figure 4.6 energy consumption of all four algorithms on the iCE40UP5K is shown. Runtimes are of the three crypto cores are identical to those used in Figures 4.4 and 4.5. It is interesting to see how little energy is used for encrypting a 64-bit block with SIMON, which is thirty times less than encrypting a 128-bit block with AES! When equating the amount of encrypted data and thus comparing two SIMON operations to one AES operation, a factor 15 in energy savings is still substantial. Saving energy on cryptography is best done by replacing standard algorithms with their lightweight versions.

Looking at the CNN, it consumes twice the energy of AES at equal runtime. If a CNN in an IMD will not be always-on but has a similar usage pattern as crypto cores, energy feasibility of FPGA neural networks in IMDs could very well be achieved.

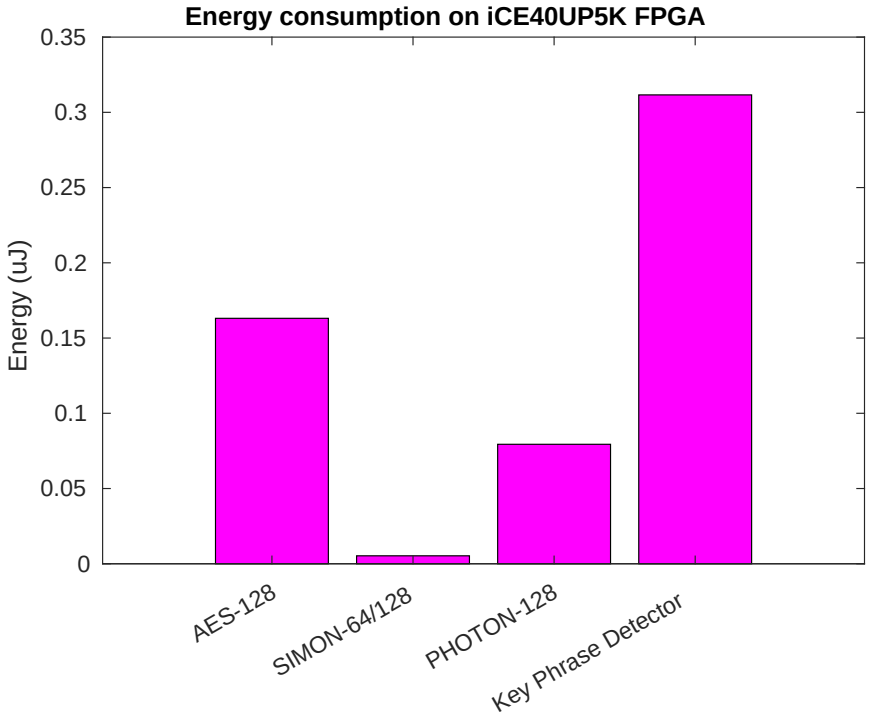


Figure 4.6: FPGA energy consumption of crypto IP cores versus Lattice CNN

IMD battery life: with or without FPGA?

So far we have only considered execution time and energy consumption of the implementations in their active state. To achieve a clear feasibility picture of an FPGA-equipped IMD, energy consumption figures have to be fitted to a realistic usage pattern. With crypto cores used for communication, it is expected that they will only be used in small, regular time intervals. For the calculations, a setup similar to [82] is taken and repeated here for completeness. The energy overhead consists of four parts: MCU, radio transceiver, electrode stimulation and cryptographic engine. It is assumed that the MCU runs at 13MHz. For the MCU overhead, a duty cycle of 5% is added like the pacemaker design in [82], with the ”Dynamic without AES” current consumption of the Tiny Gecko in Table 3.11. For static power, which can be said to have a duty cycle of 100%, the according value of sleep mode $EM4H\_VS$ is taken which equals 0.75uA. Furthermore, the energy consumption of an IMD-grade radio transceiver [83] with an effective data rate of 265 kbps is added. The duty cycle is defined as the amount of time that a peripheral is active during one day. Here, the duty cycles of the transceiver and the cryptographic engine are both set to 2 minutes of usage per day or 0.14%. Moreover, the worst-case energy consumption per heartbeat is 20uJ for commercial devices [82]. With 80 heartbeats per minute and a supply voltage of 3.3V like used in [82], the average current with a 100% duty cycle becomes 8.081uA, which is added to the total overhead. In our model, an IMD battery of 1500mAh is assumed for all battery life figures, which is typical for modern IMDs.
Two main hardware scenarios are considered:

1. Having an MCU running either software implementations or using its AES-128 ASIC.
2. Having an MCU with external FPGA, the latter one accelerating our algorithms.

Static and dynamic current draw is listed in Table 4.2. Note that especially the static current goes up dramatically when adding an FPGA, being a factor hundred higher than the Tiny Gecko alone, which is one of the main points why FPGAs are often considered too power-hungry for low-power applications. From the dynamic currents in Table 4.2, it can be concluded that all all hardware scenarios can run within the limit of 6.15mW imposed in Section 2.4. With an operating voltage of 1.2v, the upper bound for current draw is 5.125mA. thus FPGA secure authentication on harvested power is achieved.

Dynamic current gives an indication on power efficiency of each implementation, but does not give a picture on the impact of an FPGA on IMD battery life. Therefore, all data points presented in Table 4.2

Table 4.2: Static and dynamic current values for two IMD hardware scenarios

Scenario	Current draw	
	Static (mA)	Dynamic @13MHz (mA)
1: MCU (Giant Gecko)	0.00094	1.44 (SW) 1.98 (HW)
1: MCU (Tiny Gecko)	0.00075	0.58 (SW) 1.06 (HW)
2: MCU (Tiny Gecko) + FPGA (iCE40UP5K)	0.07575	3.23 (AES) 1.18 (SIMON) 0.74 (PHOTON)

are used to predict IMD battery life with the setup listed in Section 4.2.2. Resulting battery life predictions are depicted in Figure 4.7. The Giant Gecko, being a more powerful version of the Tiny Gecko, is significantly less energy efficient than the latter, barely scratching two years, where the Tiny Gecko is well into feasibility territory with its 3.76 years. It can therefore be said that having a higher performing MCU is only a good idea if recharging is available or battery life is less of a concern.
Looking at the MCU HW bars, we see that they indicate slightly less IMD battery life than with software-only operation. This is due to our model used: both run for the same amount of time, but the hardware AES ASIC will operate on significantly more data than its software counterpart.
Taking a closer look at the most interesting part of the graph, it can be seen that using an FPGA alongside a Tiny Gecko reduces the IMD battery life significantly from 3.76 to around 1.40 years, or a 63% reduction. The specific implementation that is used does not have a large influence, because the high static FPGA current is dominating the average current consumption. This static current is also the main reason for the vast drop in battery life when adding an FPGA. No FPGA feasibility of 2.5 years as defined in Section 1.3.3 is achieved here. Therefore, static FPGA power consumption needs to be mitigated to be able to obtain feasibility.

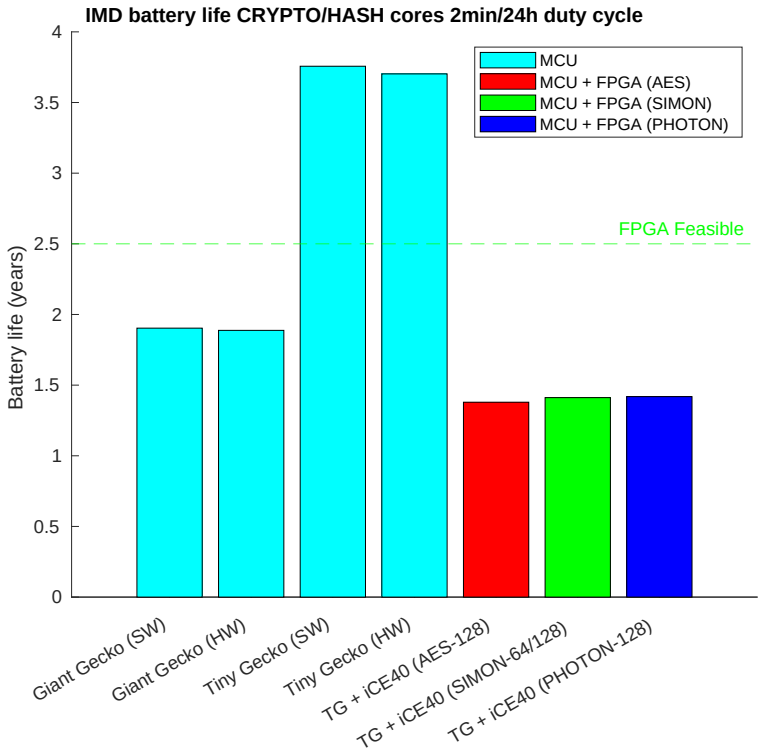


Figure 4.7: IMD battery life with and without FPGA

IMD battery life with FPGA power gating

Keeping the added FPGA in an always-on state is detrimental for IMD battery life. Hence, a strategy has to be adopted to mitigate the impact of static current on battery life. In the case of the FPGA implementing cryptography, it will only be used while communicating with a remote device. Using a duty cycle of 2 minutes per 24 hours as defined in Section 4.2.2 means that during idle time the FPGA could be powered off or put in a deep sleep state. No sleep states are available on the iCE40UP5K FPGA, but other flash-based FPGAs like the IGLOO series have a state called Flash*Freeze that result in low static currents. In our calculations, we take the Flash*Freeze current of the AGLN250 FPGA, which is comparable to our baseline FPGA, having 6144 DFFs and equivalent number of LUTs. For a voltage of 1.2V, Flash*Freeze current is 20uA [84]. In our predictions in Figure 4.8, we substitute the 75uA static current of the iCE40UP5K with the 20uA Flash*Freeze current such as to predict battery life if such a mode would exist. Next, the scenario of having external flash that saves the configuration of the FPGA is considered. This enables a complete power-off when the FPGA is not used. The red bars in Figure 4.8 showcase the three FPGA scenarios, compared to the blue MCU-only bars. Even with a lower static current in Flash*Freeze mode, the battery life already improves dramatically compared to an always-on situation, going from 1.38 to 2.47 years, thus adding more than a year of battery life! A complete power-off, which is possible with any FPGA, not just a flash-based IGLOO, results in even more gain, adding another year with 3.48 years of IMD battery life. Compared to MCU-only implementations, we see that the Giant Gecko is only more energy efficient when no FPGA power saving is available, leading to the conclusion that it is always better to use an FPGA with a small MCU than to use a higher performing MCU only, when the FPGA can be power gated while idle. Sofware-only operation on the Tiny Gecko is the most efficient, but lacks performance compared to the FPGA. Hardware acceleration, however, will only cost about 0.28 years, or 7.5% less battery life if the FPGA can be completely powered off when idle. To make an FPGA feasible in an IMD, at least 2.5 years of battery life needs to be achieved like we defined in Section 1.3.3, when no recharging is available. Therefore, having a Flash*Freeze-like mode gets us almost to feasibility for AES-128, where a complete power-off needs to be possible to obtain certainty about FPGA feasibility.

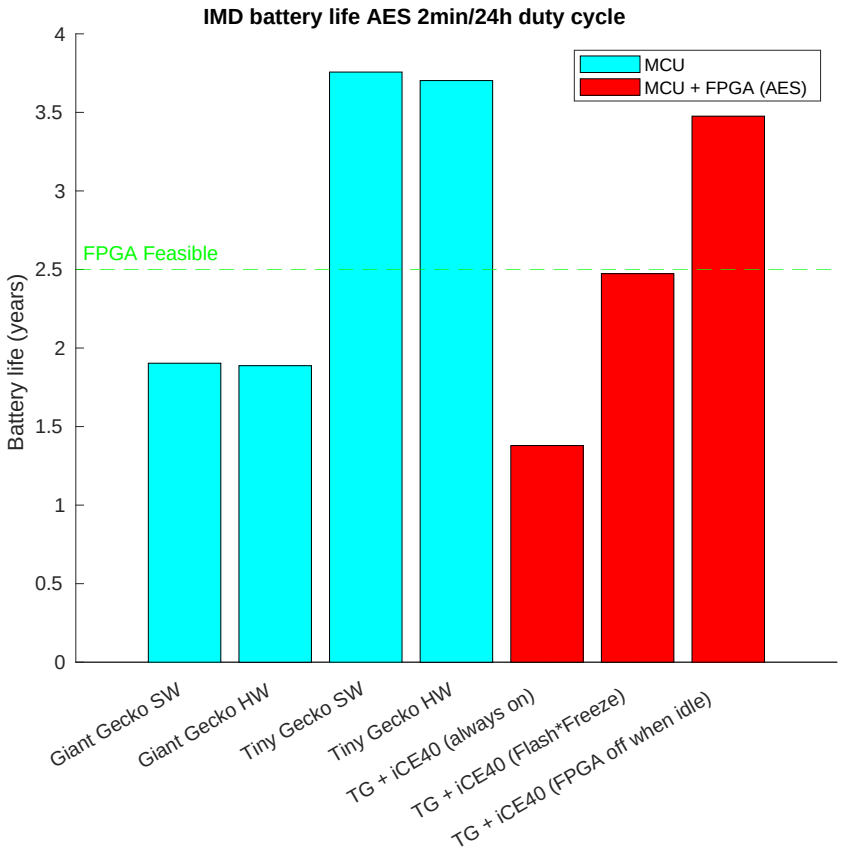


Figure 4.8: IMD battery life when power gating the FPGA

4.3. Q2: How big are the improvements of an eFPGA over an FPGA with regard to energy consumption and area?

Reconfigurable fabric is not only restricted to separate chips, but can also be incorporated with the host processor on the same die. Examples like the Zynq from Xilinx are well known, but not for their ultra-low-power focus. Furthermore, still only predefined fabric sizes are available. With the eFPGA tools from Menta, arbitrary sized eFPGA fabrics can be designed and put to test in simulation. In this section, a look is taken at the three architectures designed and described in Section 3.5.4 with a main focus on area and power. It will be shown that Menta eFPGAs can be more efficient than the Lattice FPGA in both fields.

4.3.1. eFPGA area

One of the main advantages of an eFPGA compared to an FPGA is the general reduction of physical area. This is mainly caused by the fact that an FPGA needs area dedicated to I/O interfaces and other extra logic, where an eFPGA can directly be connected to the surrounding silicon on the same die. Furthermore, eFPGA will scale down with smaller manufacturing processes, where the size of an FPGA is also limited by other factors, such as package, I/O pad size and surrounding components. In Figure 4.9 the area of Menta Architecture 1 to 3 is shown, relative to the package size of the Lattice iCE40UP5K FPGA. Note that the y-axis has a logarithmic scale. *Only the fabric and package sizes are considered here*, without taking into consideration that an external FPGA would need more PCB space for surrounding components and interconnect. Taking these factors into account, the area advantage of eFPGAs will grow even more than is depicted here. Agreements with Menta dictate that no absolute area figures may be given. Four different eFPGA process technologies were available: two 180nm X-FAB and 28nm and 7nm from TSMC. Taking a look first at the two 180nm processes, it can be

seen that they occupy substantially more area than the more modern processes of TSMC. Opting for the radiation-hardened variant makes the difference even larger: it costs 2.96 times as much area compared to the regular process. Radiation-hardened silicon is useful in environments that are prone to high radiation intensity. For IMDs, the most probable high-radiation environment is when a patient undergoes an MRI or CT scan. Although other means have been used to mitigate radiation effects on IMD hardware, such as radiation-shielding casings, a radiation-hardened process would improve radiation resistance even further.
Comparing the two 180nm processes to the iCE40UP5K, it is clear that the latter is more area efficient with its 40nm process. Differences in designed Menta architectures are not nearly as influential on the area as the choice of process technology. Seeing that a 180nm eFPGA will require twenty to hundred times more area than a regular FPGA, it can be judged that this technology family is unsuited for IMD-targeted FPGAs.
Looking at the TSMC 28nm & 7nm processes, all Menta architectures are smaller than the Lattice FPGA. Having full cryptography functionality (running AES/SIMON/PHOTON) at 12% and CNN support at 21% of the iCE40UP5K area is possible with Architecture 1 and 2 on TSMC 7nm respectively. With 28nm, the advantage is less pronounced but still there: Architecture 1 and 2 occupy 40% and 70% of the Lattice area respectively. Therefore, only the TSMC nodes are advantageous considering area. As dynamic power estimation models from Menta are currently only available for TSMC 28nm, this process is chosen in all next Menta comparisons. Generally speaking, the eFPGA area is more dependent on process technology size than its architecture. Comparison to the iCE40UP5K FPGA is mainly a comparison of process technologies and can possibly be extended by comparing the FPGA + MCU to an eFPGA-equipped MCU in area. However, due to unavailability of the latter, this comparison was not made.

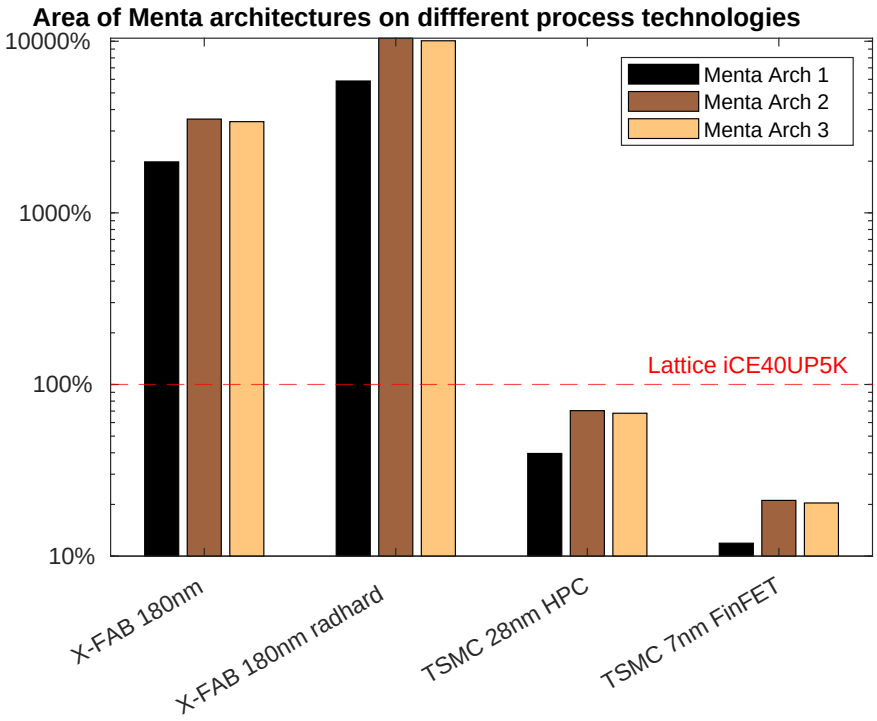


Figure 4.9: Area of Menta eFPGA architectures relative to Lattice iCE40UP5K FPGA

4.3.2. eFPGA power

Static power

While TSMC 28nm & 7nm are more area-efficient than the Lattice FPGA, static power increases with smaller process nodes. SRAM-based FPGAs, which Lattice FPGAs fall under, are known for their relatively high static power consumption. Although Menta eFPGAs do not fall in this category, using a

small TSMC node will still contribute to high static power. In Figure 4.10 the static power consumption for every process technology and Menta architecture is shown. It almost looks like the inverse of Figure 4.9, with the TSMC nodes yielding a static power consumption that is orders of magnitude higher than the X-FAB processes. Note the logarithmic y-axis here as well. Most remarkably is the difference in static power of the two 180nm processes: it is identical. One would expect more static power from a fabric that occupies more area, so the radiation-hardened variant would be more power consuming during idle. If low energy consumption is crucial and area is not a problem, the radiation-hardened 180nm variant would be the best fit in environments with high radiation intensity.
Looking at the TSMC processes, we see that the 28nm process is the greatest static power consumer. This is not only due to it being a small process as 7nm is smaller, but due to the fact that 28nm is a planar CMOS process, which is known for high gate current leakage that leads to high static power. The smaller the node, the higher the static power consumption as transistor gates get smaller and therefore leak more current. From 16nm and below, foundries are not using planar CMOS anymore but FinFETs, which strongly reduce leakage. the 7nm TSMC process is FinFET based and therefore less leaky than the 28nm process. Furthermore, the HPC in its name stands for 'High-Performance Computing', giving an indication that it is optimized for performance and not so much for ultra-low-power environments. However, as stated in Section 4.3.1, dynamic power calculations are only available for the 28nm process. With this reason, it is chosen as the process to do further evaluations of the Menta architectures on. Having a static power consumption of over 1mW, it is clear that also with our eFPGAs, a power gating strategy like in Section 4.2.2 is even more needed. This is not due to the eFPGAs themselves but because of the performance-oriented process technology. By choosing a technology that is targeted for low power, static power can be reduced by 10 to 100 times as is shown in Figure 4.10.

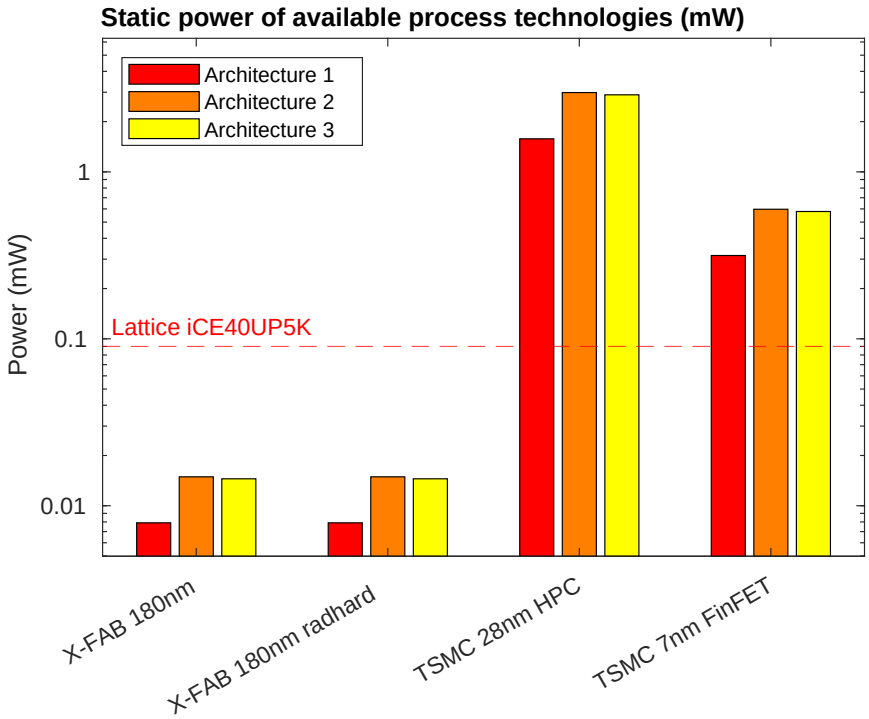


Figure 4.10: Static power of Menta architectures on all available process technologies

Total power

To get a complete picture on power draw of all reconfigurable fabrics, dynamic and static power are added together and displayed in Figure 4.11. The reported dynamic power parts of the iCE40UP5K FPGA are obtained by multiplying the current measurements described in Section 3.6.2 with the operating voltage of 1.2V. Unlike the iCE40UP5K, the Menta architectures cannot be measured as they do not physically exist, but power is estimated with the help of a model provided by Menta. Note that, as stated in Sections 4.3.1 and 4.3.2, all Menta power calculations are based on the architectures using

the 28nm HPC process from TSMC. The general trend observed here is that static power increases when process node size and dynamic power decrease.No average activity factor for implemented algorithms could be determined as stated in Section 3.6.3, so multiple activity factors have been assumed, making up for differences in switching activity for different designs. Figures of 10%, 20% and 30% are used, corresponding with typical figures from designs from industry. 12.5% is reported by Menta as a typical figure of their customers, while a common switching-intensive design has activity around 20%. Therefore, the 30% entry can be seen as a worst-case scenario. However, even with such a high activity factor, the total active power is less for running the Key Phrase Detector project on Architecture 2 and 3 than it is on our baseline FPGA. Likewise, running AES is more efficient on Architecture 1, which is specially designed for security primitives, than on the Lattice platform. Again, a smart power gating strategy that eliminates the major part of static power consumption will result in eFPGAs to be the better choice, looking at active power consumption.

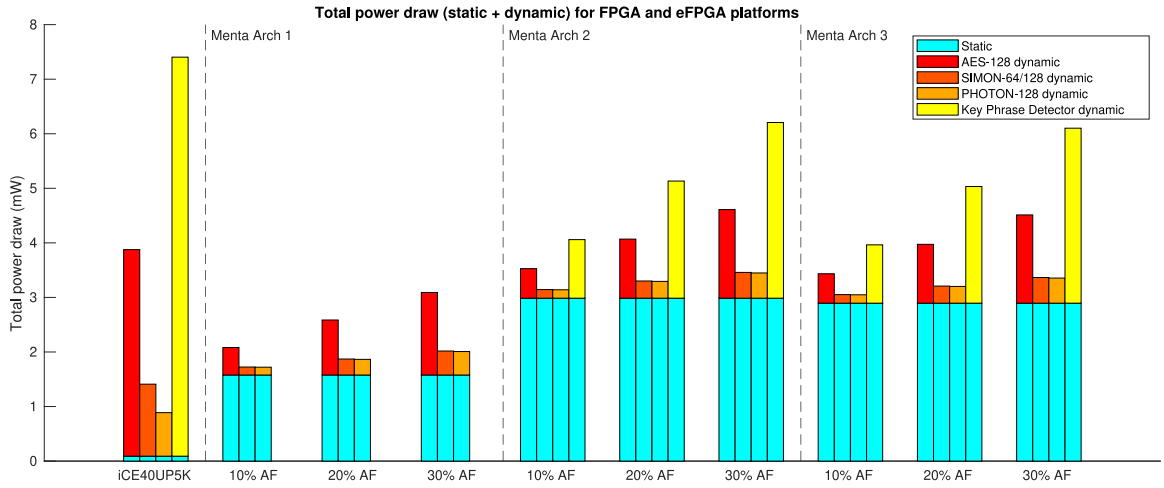


Figure 4.11: Total power of Menta architectures on all available process technologies

4.3.3. CNN-equipped FPGA vs eFPGA battery life

So far we have seen battery life analysis of an IMD with and without external FPGA running cryptography in Section 4.2. As has been said, security primitives will only be active for a very limited amount of time during the day. With artificial neural networks, this will most likely change. Take seizure prediction for example, a field in which neural networks could be deployed for pattern detection. Here, the neural network would have to continuously monitor its input to be able to be successful in seizure prediction. A similar behavior is observed in the Key Phrase Detector project: the CNN inside is continuously scanning its inputs for a spoken key phrase. With this in mind, energy calculations have been done for multiple duty cycles and continuous operation. The results can be found in Figure 4.12, where battery life of an iCE40UP5K-equipped IMD is compared with one that has Menta eFPGA Architecture 2 or 3 incorporated. Each bar in the graph is cumulative, meaning that the upper colored part also includes the parts below. As is clear from Figure 4.12, the most influential factor on battery life is not the type of eFPGA used, but its duty cycle or usage pattern. Using our baseline FPGA is less efficient with smaller duty cycles and a power gating scheme, which is logical since dynamic power is dominating there. Getting into the feasibility range of 2.5 years of usage, a duty cycle of 2 minutes with at least a Flash*Freeze-like mode is needed. If one of our eFPGA architectures is used and the reconfigurable fabric can be turned off when idle, the duty cycle can be widened to 6.5 minutes per day. In our case, eFPGA fabric will result in a longer battery life than the traditional FPGA. But, a ceiling of a couple minutes per day makes it seem that continuous monitoring is out of the question. However, some neurostimulators currently on the market feature recharging capabilities and ask the user to recharge the device regularly [85]. If recharging is available, even continuous usage could be allowed, seeing that in all three cases the battery life will be 10 to 12 days. Therefore, with recharging, running a hardware-accelerated CNN in an IMD is definitely feasible.

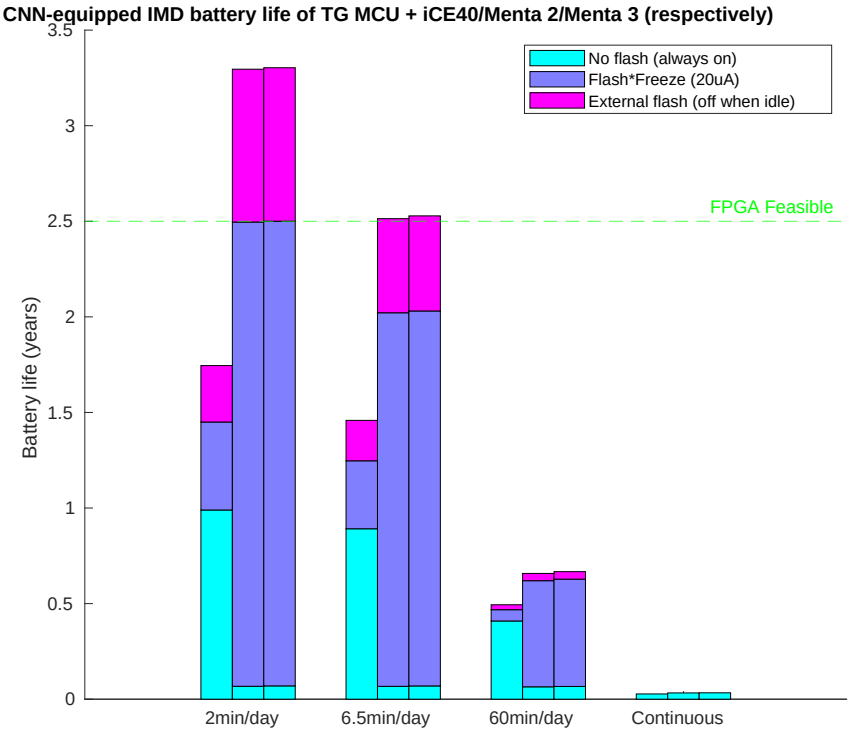


Figure 4.12: Battery life of an IMD with neural network-equipped (e)FPGA. For the eFPGA platform, a 30% activity factor has been used

4.4. Q3: In what cases is daily FPGA reconfiguration beneficial in IMDs?

Until now, we have considered FPGAs in IMDs that have already been configured with a bitstream and will only consume energy when operating (dynamic) or idling (static). One of the unique features of FPGA fabric is the ability to reconfigure the hardware so that new functionality can be implemented without changing the physical hardware of the IMD. This is also possible with software, but as we have shown in Section 4.2, it is preferred using hardware acceleration for cryptography and neural networks when performance and energy consumption are considered. Reconfiguration opens a realm of possibilities: an FPGA fabric could implement a neural network for seizure prediction, and only during time of communication reconfigure itself to a crypto core. Swapping the FPGA configuration back and forth once a day is the case investigated in this section. Multiple hardware accelerated algorithms could be supported by the same fabric. But, two major factors determine if daily reconfiguration is feasible: its latency and energy consumption. Feasibility in terms of latency is achieved if execution time of crypto cores, including reconfiguration, is still less than the total execution time if run in software. A look is also taken at energy consumption including reconfiguration for AES, to see what the energy cost is of frequent reconfiguration. With these results, it can be determined in what cases it would be beneficial to have an FPGA in an IMD, regardless of the added cost of reconfiguration.

Our configuration case

In answering this question, it is assumed that we have an FPGA running a functional algorithm that provides medical therapy. During one moment of the day, the FPGA is reconfigured to function as hardware-accelerated crypto peripheral, using one of our three security primitives listed in Section 3.2. This is to simulate the daily data transfer from and to the base station belonging to the IMD. After the data has been encrypted and transferred, the FPGA is configured back to its original application. For execution time and energy consumption, this means that delay and energy of two configurations will

be added to the FPGA results. The iCE40UP5K FPGA platform will be used as reference and no eFPGA considerations will be made here, as sufficient data on reconfiguration of Menta eFPGAs was not available to us.

4.4.1. Configuration latency

Determining configuration time

Regarding the time needed for reconfiguration, we take the iCE40UP5K platform as reference. Three configuration speeds are supported, only differing in the operating clock speed used. The slowest and default mode uses a 12MHz system clock. Assuming that in our FPGA-equipped IMD only a single clock domain is present, this mode is used in our calculations. Next to the configuration speed choice, the iCE40UP5K FPGA has two modes in which the FPGA can be reconfigured. In the first mode, **SPI Master**, the device configures itself either from its one-time writable Non-Volatile Configuration Memory (NVCM) or from external SPI Flash. No MCU or external programmer is needed for this process. In **SPI Slave** mode, the FPGA is configured by an external processor acting as a programmer. Both modes could be used in an IMD, where the choice would mainly depend on resource efficiency and physical topology of the IMD hardware. Both modes are similar in the way they transfer a configuration image to the FPGA, namely bit sequential, one bit at every SPI clock cycle without interruption.
The time needed for a single FPGA reconfiguration can be calculated from the iCE40 Programming and Configuration manual [86]. The only official figure for our configuration speed stated in the iCE40UP family datasheet [87] as ”SPI Master or NVCM Configuration Time” is 140ms. However, it is not clear how this number is derived nor if it is also applicable to the SPI Slave mode. Therefore, own calculations have been performed with Equation (4.1) for Master and Equation (4.2) for Slave mode, based on Figure 9.4,9.5 and 13.2 of the Programming and Configuration manual [86].

$$t_{SPI\_Master} = T_{clk} * (cmd_{fastread} + cmd_{startaddr} + dummy_{start} + bitstream + cmd_{powerdown}) \quad (4.1)$$

$$t_{SPI\_Slave} = T_{init} + T_{memclr} + T_{clk} * (dummy_{start} + bitstream + dummy_{end}) \quad (4.2)$$

Appropriate values have been selected for the various variables present in Equations (4.1) and (4.2). Our configuration speed fixes T_{clk} at 12MHz. T_{init} is a fixed time of 200ns needed to enter Slave mode, where T_{memclr} is 1200us, during which the configuration memory of the iCE40 FPGA is erased. The other parameters are counted in number of cycles. $dummy_{start}$ and $dummy_{end}$ are 8 and 149 cycles respectively. $cmd_{fastread}$ is an 8-bit sequence which indicates to the external Flash memory that a read operation is performed and is sent right before $cmd_{startaddr}$, which contains a 24-bit start address. Both are sent sequentially in 32 cycles. At the end of an SPI Master configuration, the external Flash may be powered down with the 8-bit $cmd_{powerdown}$.
Regarding the most important and substantial part of Equations (4.1) and (4.2), the bitstreams have a length only depending on the FPGA model and not on the configuration that is loaded. For the iCE40UP5K FPGA, the total bitstream size equals 833288 bits. Filling in Equations (4.1) and (4.2) and multiplying by 1000 to get the answer in milliseconds instead of seconds, we get the respective configuration times in Table 4.3 from Equations (4.3) and (4.4):

$$t_{SPI\_Master} = [\frac{1}{12 * 10^6} * (8 + 24 + 8 + 833288 + 8)] * 1000 = 69.44ms \quad (4.3)$$

$$t_{SPI\_Slave} = [200 * 10^{-9} + 1200 * 10^{-6} + \frac{1}{12 * 10^6} * (8 + 833288 + 8)] * 1000 = 70.64ms \quad (4.4)$$

Table 4.3: iCE40UP configuration time for SPI Master and Slave modes

Configuration mode	Frequency (MHz)	Configuration time (ms)	Average current (mA)
SPI Master	12	69.44	-
SPI Slave	12	70.64	0.17

Due to the bitstream size being the most determining factor in the configuration time, both modes result in about equal configuration time. In our calculations, the worst-case configuration time is taken, which is the SPI Slave time of 70.64ms. The accompanying current draw of 0.17mA in Table 4.3 was obtained by measurement on the iCE40UP5K-B-EVN board during a configuration procedure. As a double reconfiguration scenario is simulated as described in Section 4.4, the added time delay for daily reconfiguration will be **141.28ms**.

MCU to FPGA with reconfiguration comparison: execution time

With the added time delay for reconfiguration quantified, it is possible to compare execution times of our crypto cores on our baseline Cortex-M0+ MCU to the FPGA, done likewise in Section 4.2.1. But, seeing that both block ciphers execute encrypting a single block in less than 4ms and that even a PHOTON software run on one block finishes faster than a double reconfiguration (81ms vs 141.28ms), it can already be said that daily reconfiguration is infeasible compared to software when only having to encrypt a single 128-bit data block. Having that the reconfiguration delay penalty always occurs once and never more than once, it follows that using the encryption engine for a longer time is needed to unlock the potential of the FPGA. Therefore, data messages of different sizes, ranging from a single 128-bit block to 32kiB (256 blocks) are used in our calculations. The results are depicted in Figure 4.13. Note the logarithmic scale on the y-axis.

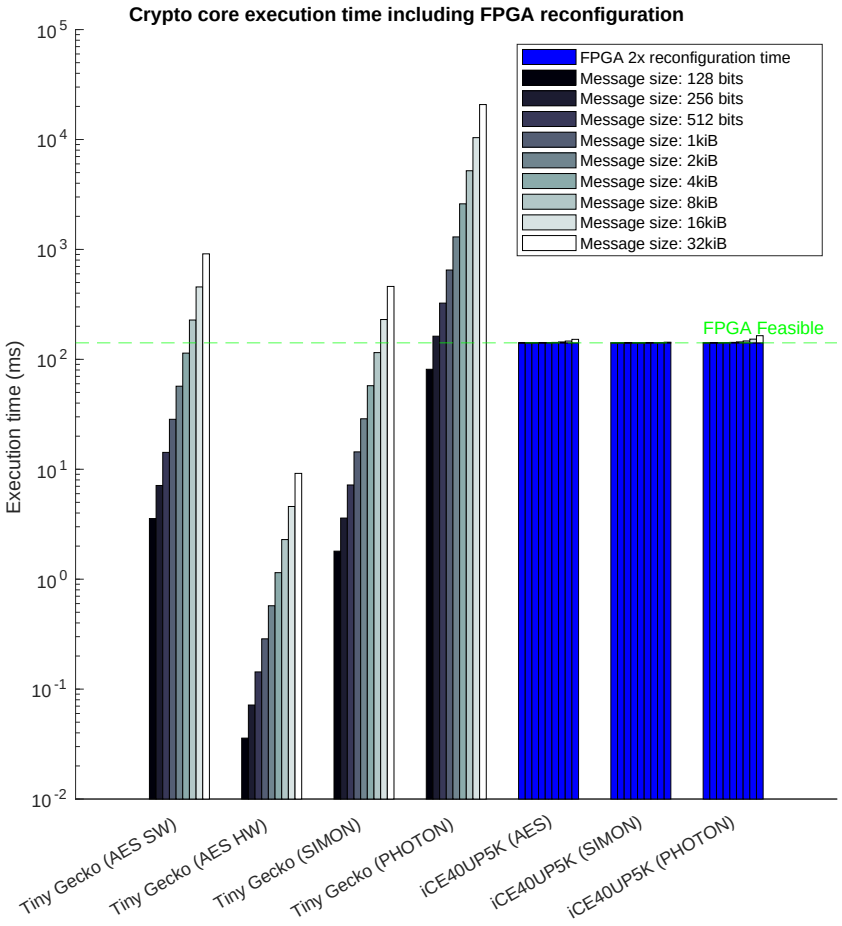


Figure 4.13: Execution time of all crypto cores, considering FPGA reconfiguration

When comparing AES runs on the MCU with hardware acceleration enabled against the FPGA, it is already clear that an FPGA lags ever further behind with added reconfiguration latency. There will never be a point when the FPGA is faster than the HW-accelerated MCU, as encrypting a single 128-bit block is faster on the MCU while the FPGA already starts with a reconfiguration penalty.
Regarding the software implementations, there is a tipping point for each algorithm where the FPGA starts to be faster. Note that the reconfiguration time is the major contributing factor and that the respective algorithm run on the FPGA is less relevant. The most resource efficient algorithm, SIMON, is feasible to run from messages 16kiB or larger. Taking the data rate of 265kbps assumed for our transceiver [83] in Section 4.2.2, the amount of data that can be transferred per second is slightly more than 32kiB. With our 2min/24h duty cycle of both the cryptographic engine and transceiver, it is safe to

say that the FPGA-advantageous situations will most likely be reached in real-world scenarios.
Moreover, the advantage grows when comparing to AES, for which the FPGA becomes profitable from messages of 8kiB. With PHOTON it is better to use an FPGA starting from a mere 256 bits, where it can be advised to always use an FPGA in this case.
Considering execution times, even with an added daily reconfiguration penalty it is always profitable to replace a software implementation with an FPGA. ASIC-accelerated implementations are faster but lose the reconfigurability of FPGAs, making the latter viable candidates for hardware-accelerated cryptography in IMDs, even with daily reconfiguration.

4.4.2. Configuration energy

Having proved that FPGAs are feasible and beneficial in IMDs with reconfiguration regarding execution time, it remains to show that feasibility is also confirmed from an energy point of view. The same scenario as described in Section 4.4 is used here, with energy required for double FPGA reconfiguration added to the security primitive energy. Only AES is represented here, as this makes the graph less cluttered and it is the only algorithm to have MCU with and without HW acceleration and FPGA benchmark results available, enough to draw a feasibility picture.

Determining configuration energy

For determining how much energy is consumed during configuration, we take the calculated double configuration time of 141.28ms in Section 4.4.1 and multiply it with the average current draw during the configuration phase. As no data is available on current draw in the FPGA datasheets, the FPGA current was measured during configuration from a PC in the way described in Section 3.6.2. Current values between 0.15mA and 0.19mA were observed during Slave SPI configuration at 12MHz. The average of these values is taken, resulting in 0.17mA configuration current. Knowing that the configuration clock speed is 12MHz and taking the default of 1.2V as FPGA operating voltage, the resulting double configuration energy becomes **28.82uJ**.

MCU to FPGA with reconfiguration comparison: energy

Using the FPGA is deemed feasible when the consumed energy including double FPGA configuration is less than the same operation executed in software. For this end, the energy consumption of AES encryption for MCU software, MCU HW accelerated and FPGA is depicted in Figure 4.14. Note the split y-axis with two different scales. The same encryption message sizes of Section 4.4.1 are used here. Running AES with MCU HW acceleration is always more energy efficient that on an FPGA, but this is again to be expected. However, the execution time tipping point for AES compared to software was at message sizes of 8kiB, but for energy it already occurs at 1kiB! FPGA feasibility therefore is achieved faster looking at energy than with execution time. As mentioned in Section 4.4.1, the expected amount of data transfer between the daily FPGA reconfigurations is well over 32kiB, proving that an FPGA in an IMD will also be feasible regarding energy consumption, even making a stronger case than execution time.

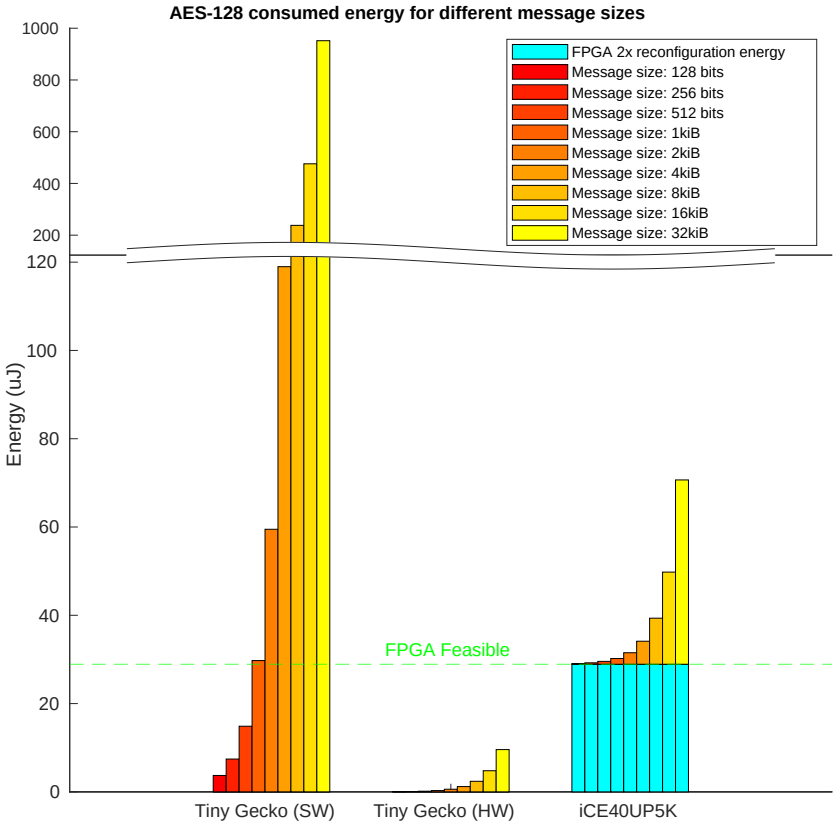


Figure 4.14: Energy consumption during AES encryption, considering FPGA reconfiguration

4.5. Q4: Having an FPGA-equipped IMD, what are the difficulties in aspect to legal certification?

Before an IMD can be implanted in a patient to perform its intended therapy, it has to pass certification according to Medical Device Regulation 2017 [88] for the European region, or via the Food and Drug Administration (FDA) in the USA. Requirements on what is needed to pass certification depends on the type of product being under consideration. In this section, we will mainly focus on the set of regulations imposed under the MDR2017/745 regulation which is in effect for Europe and first derive how our category of interest, implantable medical devices, is classified. With this information, the conditions and rules under which IMDs fall can be determined, especially those related to software and FPGA updates. Knowing which rules have to be followed for FPGA-equipped devices, we can draw conclusions on the additional legislative steps to be taken for these devices, or conclude that existing regulation does not impose additional difficulty.

4.5.1. Classification of IMDs according to MDR2017/745

Not only IMDs, but all products that have a medical purpose are to be certified. Depending on the product and its purpose, a class is assigned which mainly determines the rigorousness of certification applied. In Figure 4.16 the different classes are depicted, ranking from Class I to III according to increasing risk for the patient. In the MDR2017/745 regulation, there are 22 rules listed in Annex VIII that determine in what class a medical device falls. A summary of these rules is presented in Figure 4.15, retrieved from easymedicaldevice.com. Only devices in Class I can be self-certified and do not require an external agency, indicated as Notified Body in Figure 4.16, to review the certification

procedure. An example of this is a wheelchair, that has a safety test report included in its manual. Generally speaking, three factors are influencing the classification of a medical device:

1. **Device type**, being Non-invasive, Invasive, Active or under a special rule
2. **Risk involved**, following that life-critical applications get assigned a higher class than life non-critical ones
3. **Duration of use**, which can be **Transient** (less than 60 minutes), **Short term** (between 60 minutes and 30 days) or **Long term** (more than 30 days).

To arrive at the final classification of a device, the highest resulting classification from any rule that applies to it determines the class. For IMDs, it is clear that they have to be classified as Active or Invasive devices. Pacemakers and neurostimulators are classified as Class III by rule 8, which states that all devices which are intended to be used in direct contact with the heart, the central circulatory system or the central nervous system are classified as Class III. Any other rule could also apply to these IMDs, however, with rule 8 already indicating the highest class, no other rule needs to be evaluated. Regarding duration of use, IMDs are implanted and as such always aimed at long term use. Risk level translates to the class of a device, with Class III indicating that IMDs are high-lisk devices. The life-criticality does not influence the classification any further, this is only affecting classification on a device that is of itself lower than Class III.
Although already having determined the class of IMDs, the other rules are still interesting to examine to see whether mention is made of any software or hardware reconfigurability, our field of interest. Rule 11 is about software, but considers only software running on computers or other external monitoring devices and thus is not relevant for IMDs. Rule 22 is about active therapeutic devices with an integrated diagnostic function which significantly determines the patient management by the device. Neurostimulators or monitors can also fall under this rule, however no mention is made of any software part. Therefore, other parts of the MDR2017/745 regulation document have to be consulted for a clearer idea on the position of functionally upgradable IMDs.

4.5.2. Regulations considering hardware and software in IMDs

Now that we know that IMDs fall in Class III, a next point to examine is whether regulations exist considering Class III devices that can change their functionality with a software or hardware update.

Hardware configurations

Searching for 'hardware' gives only three hits in the complete MDR2017/745 document. Two of them consider minimum hardware requirements for PC software and are as such not relevant to our research. The third occurrence is in section 6.1 of Annex II, where it is listed that a number of test report details are required to present at verification. Quoting 6.1 (b), on required information supplied at verification: *software verification and validation (describing the software design and development process and evidence of the validation of the software, as used in the finished device. This information shall typically include the summary results of all verification, validation and testing performed both in-house and in a simulated or actual user environment prior to final release. It shall also address all of the different hardware configurations and, where applicable, operating systems identified in the information supplied by the manufacturer)*. A number of points can be taken from this snippet. First, an extensive report that makes clear that rigorous software testing was done needs to be supplied. Second, all hardware configurations need to be listed on which the software runs. For traditional MCU-equipped IMDs that only run software, all used MCUs and software programs need to be evaluated and present in the report. As for FPGAs, this snippet could imply that all FPGA bitstreams need to be individually approved as they count as different hardware configurations, but no explicit mention of IMDs, let alone FPGAs, is made here, making our judgment far from final.

Configurable and software devices

Looking at Annex VI, which is about assigning identifiers to medical devices, section 6.4 (Configurable devices) and 6.5 (Device Software) spike particular interest.
Section 6.4 indicates that a unique device identifier (UDI) shall be assigned to the configurable device in its entirety as the configurable device UDI. Furthermore, an identifier is given to groups of configurations and not per configuration within a group. For our situation, this could be interpreted as having to

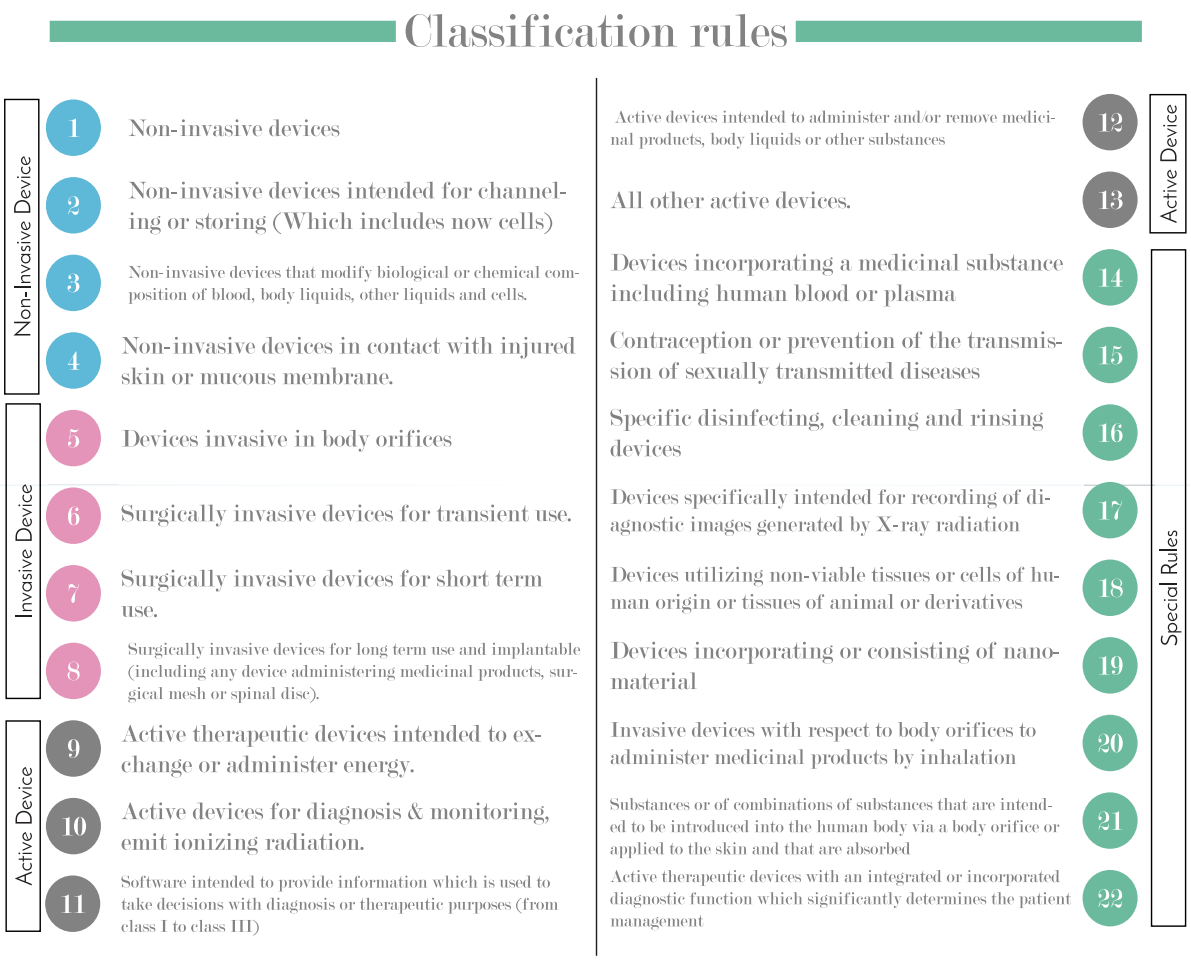


Figure 4.15: Rules for classifying medical devices according to MDR2017/745, Annex VIII regulation [89]

assign only a single identifier to the group of crypto IP cores, while requiring another one for the ANN implementation. But, no further precision is given on the word 'configuration', so it can be assumed that a device with different physical configurations is meant here.
Section 6.5 states that software that stands on its own requires a separate UDI, indicating that it is seen as a complete medical device. For software that is part of a medical device, as is the case with IMDs, one identifier will identify the whole medical device, with hardware and software components. This is consistent with section 3.3 of Annex VIII, which states that *"Software, which drives a device or influences the use of a device, shall fall within the same class as the device"*. Hereby it can be concluded that any IMD software falls under existing regulation. But, because of the volatile nature of software, it is thus interesting to see what effect these rules have on re-certification.

Re-certification
As changes in software or hardware configuration could imply new functionality, re-certification of the device most likely needs to happen depending on the impact of the change. According to section 4.11 of Annex VII, re-certification needs to happen at least every five years, where all changes since the last certification need to be taken into consideration. Again, no mention is made explicitly of changes in IMD configurations in that period, so the conditions on which re-certification need to happen are not yet clear from this regulation.

In conclusion of the MDR2017/745 document, regulations that target Class III IMDs do not discuss reconfigurable hardware like FPGAs. Regulations that come the closest to our IMD cases are about

software, either as a device itself or incorporated into an IMD. Therefore, IMDs with MCUs running software are fully covered under existing regulation and pose no novel difficulties in obtaining certification. This is further proven by the fact that MCU-equipped devices with software upgradability are already certified and on the market [2]. This situation therefore begs the next question:

Can FPGAs be seen as software from a regulatory point of view?

The next sections will answer this question with the help of resources outside of the official MDR2017/745 regulation document.

4.5.3. Considering FPGAs as software under regulation

As we know from computer engineering, FPGAs are mostly put under hardware solutions as opposed to software running on CPUs. However, from a regulatory point of view, FPGAs tend to have the same characteristics as software, sharing ease of updating functionality and fixing bugs. With the help of external sources, we will verify the hypothesis that FPGAs in IMDs fall under the same regulations as IMD software.

External sources

According to Blue Pearl Software, a company specialized in ASIC and FPGA verification, *"all integrated circuits are generally considered as black-boxes for regulatory purposes, including ASICs and FPGAs. However, in systems where the safety function is highly dependent upon the ASIC/FPGA functionality, the detailed design of that device may become a central part of the safety assessment"* [90]. To translate this statement to FPGA-equipped IMDs, it will depend on the functionality of the FPGA whether further knowledge of its executed algorithms is needed by regulatory bodies. "Safety" in the Blue Pearl Software document means the architectural mechanism, here called safety function, that handles hardware or software faults caused by bugs or design errors, such as deadlock in a malformed FSM. It does not refer to the life-criticality of the device. If we look at our two main FPGA use cases as defined in Sections 1.3.1 and 1.3.2, wireless communication with a base station is generally considered a luxury addition in newer IMDs, not a vital part of the safety function of the IMD. Although the second case may possibly be more life-critical than the first, still no safety function is to be executed by FPGA fabric. This leads to the assumption that our FPGAs can be considered black boxes under regulation, which means that certain rules will still apply, but no knowledge is needed of the internal design by the regulatory body.

Having made sure that an FPGA can be certified under current regulation, the requirements and regulations that are relevant can differ depending on the complexity of the design. Blue Pearl Software distinguishes three categories (quoted from [90]):

1. If the ASIC or FPGA is purely based on RTL, then you may only need to apply IEC 60601-1 - Medical electrical equipment - Part 1: General requirements for basic safety and essential performance.
2. If the ASIC or FPGA has a processor that runs a small amount of C code that did not require software architectural design, both the hardware and software may be considered as a black box and only IEC 60601-1 is required.
3. If the ASIC or FPGA is a System-On-Chip, then apply IEC 60601-1 if the program portion is tiny enough, otherwise apply IEC 62304.

We see here that two standards are considered: IEC 60601-1 [91] which regards medical electrical equipment in pure hardware designs, and IEC 62304 [92], which considers more complex designs. The main factor for deciding which to use is the complexity of the design: if the FPGA design is simple enough that all its states can be tested and all edge cases evaluated, it can be considered a hardware design, only needing to follow IEC 60601-1. But, for more complex designs that, just like software, cannot be tested for al of its input-output combinations, IEC 602304 has to be followed. Our use cases consider FPGA implementations that fall certainly in the second category, being complex enough that the existing software regulation has to be applied to the FPGA.

In addition, an expert on embedded software for medical devices [93] affirms that IEC 62304 is almost certainly needed for FPGAs and warns against trying to 'loophole' certification by getting a complex FPGA design classified as pure hardware. Furthermore, the categorization stated by Blue Pearl Software is confirmed here in a similar fashion, stating that the design complexity determines the regulation

standard, only 'avoiding' IEC 62304 for the simplest designs. A similar viewpoint is conveyed by a National Instruments employee specialized in medical certification [94], stating that FPGAs are easier to test and get approval on because of the nature of hardware versus software designs, with the latter tending to be more prone to bugs undiscovered at production phase. However, companies trying to find said loophole are discouraged in doing so as almost every design can have faults, needing proper regulation.
On the other end, IEC 62304 got updated to prevent this loophole from being used as stated in the changelog for the 2015 version: *"Scope – Clarified definition of what software is (any instruction that runs on a processor). This is intended to close the perceived loophole for FPGA source code and microcontroller firmware. Note that FPGA source code is now subject to the SW life cycle requirements of the standard. (Section 1.2)"* [95], a further affirmation of FPGAs falling under the same regulations as software. To remove all remaining doubt on regulating FPGAs as software, the FAQ supplied with EN 62304:2006 [96] states unequivocally that *"Software executed on a processor (can also be part of a FPGA) during the intended operation is considered a software item under EN 62304"* in section 2.1.4 b).

Conversations with certification experts
Our preliminary conclusion that FPGA designs fall under existing software regulations is confirmed by two certification experts to whom we spoke directly.
Monir El Azzouzi from easymedicaldevice.com indicated that whether or not an FPGA is used for a certain functionality, this hardware part is only a tiny part of the complete verification picture. Important is to answer the user needs and designing the medical device around those needs. After the device has been classified according to existing rules, it can be allowed on the market, with no significant distinction in regulation between software and hardware-based designs in IMDs.
Achilleas Tsoukalis of Micrel Med gives further elaboration on the re-certification of FPGA-equipped medical devices. First, as the FDA treats components like FPGAs as black boxes, meaning that the specific hardware is not considered in verification, only its function. The same is true for MCUs, making no distinction between hardware and software implementations. If the component in question is not life-critical and proper black box verification is done, the FDA approves the device. If it is life-critical, more rigorous verification is needed but there is again no difference between software and hardware devices. For minor edits like bug fixes or algorithm tunings, it is not even necessary to wait on re-approval. The company needs to inform the FDA of the change but no complete re-certification is needed.

4.6. Conclusions

In this chapter, our results answering the four research question and problem statement are discussed. Based on resource usage, only one data point is taken per security primitive, as the UART and AHB-Lite interfaces have a similar resource footprint. The largest resource usage impact is made by the algorithms: choosing SIMON instead of AES as block cipher makes up for a factor 3.5 decrease in resources used. Due to the Menta architecture being LUT6 based rather than Lattice's LUT4, calculations were done to extract the supposed resource usage of the Key Phrase project on Menta fabric, which was found to be 75% that of Lattice. Seeing that all DSPs are used in the Key Phrase Detector implementation, it may be DSP limited, which can be alleviated in the Menta architectures by placing more DSPs. This could potentially result in less resource usage than calculated here. With this figure, the size of eFPGA Architecture 2 and 3 was determined.
Compared to the AES ASIC present in our MCUs, the FPGA implementation is 17% to 40% slower, but FPGAs can be used for retaining speedup over a wider variety of algorithms. Block ciphers in software are tolerable but hashing with PHOTON-128 is not, taking 81ms to hash 128 bits with the Tiny Gecko. The FPGA will complete this task in a mere 1073 cycles, or less than 0.1ms. Therefore, FPGAs are beneficial in IMDs considering performance while retaining functional flexibility.
But, the FPGA consumes 4.4x the energy compared to our MCU with AES ASIC, although this can be mitigated by employing a custom, more lightweight cipher. Replacing AES with SIMON results in an energy consumption reduction of more than 11 times. Next to cryptography, FPGA hardware accelerated neural networks are possible with only twice the energy consumption as AES under the same runtime.
Adding an FPGA to our baseline Tiny Gecko MCU results in a 63% drop from 3.76 to 1.40 years, rendering the addition unfeasible. However, with static FPGA current being the major contributor to average

power consumption, power gating provides a solution. We simulate our iCE40UP5K FPGA having a power efficient sleep mode, like Flash*Freeze from Microsemi IGLOO devices, as well as turning the FPGA off completely when idle. These scenarios account for a jump from 1.40 to 2.47 and 3.48 years respectively, resulting in the feasibility of FPGAs in IMDs considering battery life, with a decrease of only 7.5% in the last scenario.
Considering physical area, using TSMC 7nm the Menta eFPGAs are only 12% and 21% the area of the iCE40UP5K for cryptography-only and neural network support, respectively. Using a 180nm process results in an area of 20 to 100 times that of the Lattice FPGA, which is unfeasible. The selected technology for all further experiments is TSMC 28nm, which results in 40% to 70% of the Lattice FPGA area, making the eFPGA more area-efficient than the baseline FPGA. Only chip and fabric area is considered without surrounding components and interconnect in the case of the baseline FPGA, which would skew the area advantage even more in the direction of the eFPGAs. The eFPGA area is more dependent on process technology size than its architecture. Comparison to the iCE40UP5K FPGA is mainly a comparison of process technologies and can possibly be extended by comparing the FPGA + MCU to an eFPGA-equipped MCU in area. However, due to unavailability of the latter, this comparison was not made.
The battery life expectations of FPGA and eFPGA scenarios show that even with a very high modeled average activity factor of 30%, the eFPGAs are more efficient than the iCE40UP5K FPGA. Due to their low power consumption, operating the security primitives on harvested power is possible, with no recharging needed. When recharging every week is available, continuous neural network usage is feasible as expected battery life is 10 to 12 days. Otherwise, only 6.5 minutes active usage per day for either eFPGA architecture is the maximum to retain energy feasibility.
To determine if daily FPGA reconfiguration is feasible in IMDs, a scenario was set up where an FPGA is configured to a cryptographic engine and operates for two minutes, after which the FPGA is configured back to its original state. Two parameters determine the feasibility: execution time and energy consumption added with a daily double reconfiguration. Regarding energy, it was found that the FPGA is faster than software AES, SIMON and PHOTON starting with messages of 8kiB, 16kiB antd 256 bits respectively, with an added 70.64ms configuration penalty on the FPGA side. The feasibility case was even stronger considering energy: software AES was at messages from 1kiB already less efficient than the FPGA with an added configuration cost of 28.82uJ. Within our use case, these message size thresholds will easily be reached. This renders FPGAs feasible in IMDs with both performance and energy configuration costs added.
No direct mention is made of FPGAs in European regulation, but multiple sections state regulations for software, which is therefore fully covered. Consulting certification experts, it was found that integrated circuits are generally considered as black boxes in regulation, meaning that only its functionality needs to be verified, but that it does not matter if an MCU or FPGA is inside the box. If verification of FPGAs can be brute forced by exhaustive testing of all possible input and output combinations, it is a hardware device falling under IEC 60601-1. Otherwise, IEC 62304 needs to be applied, which is the same regulation as for software components. This distinction is universally held by regulation companies and experts in certification. Therefore, regulation will be the same as with state-of-the-art MCU-based IMDs with no additional legal difficulty in place.

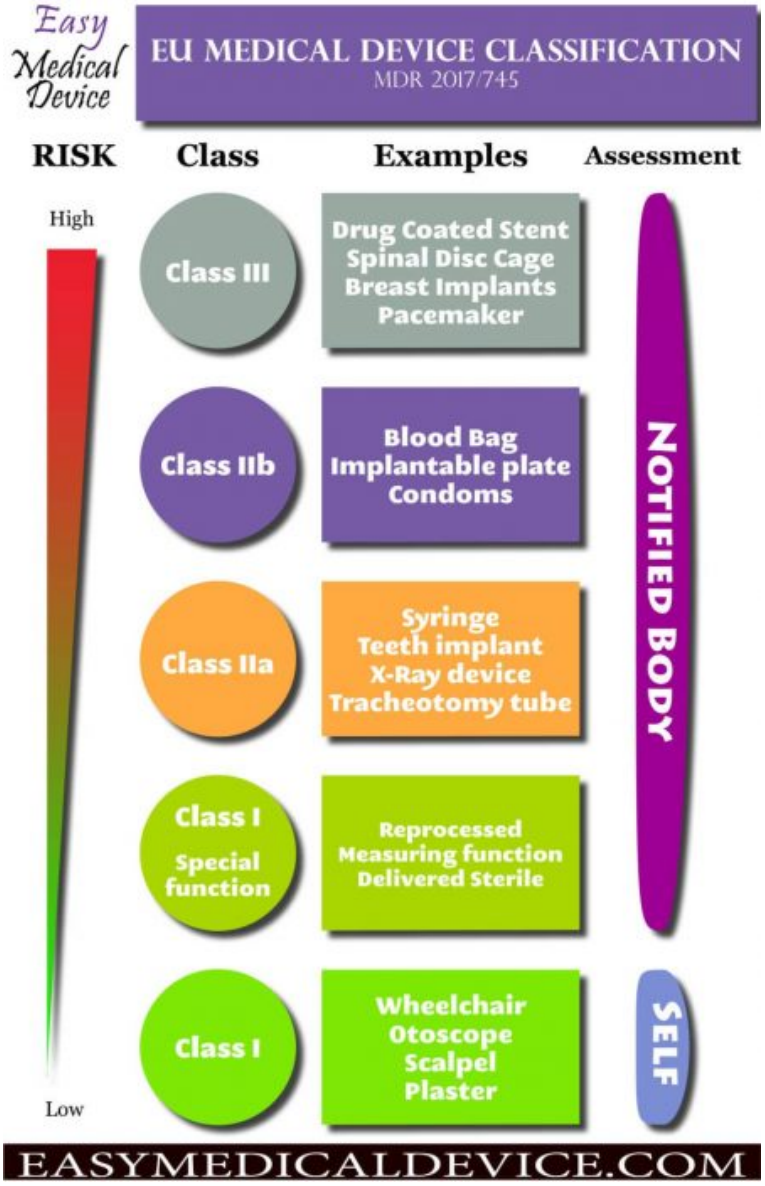


Figure 4.16: Classes of medical devices according to MDR2017/745 regulation [89]

5 Conclusion

5.1. Summary

In Chapter 2, we gained background knowledge on the following four subjects: Ultra-low-power FPGA and eFPGA technology, cryptography for ultra-low-power hardware, artificial neural networks and their application in low-power environments, and wireless energy harvesting.
FPGAs have become a viable option for the ultra-low-power domain with recent technology developments. They have been proposed for IMDs in literature, however, no extensive energy analysis on FPGA-equipped IMDs has been done. To be able to answer our first and second research question and contribute to this knowledge gap in literature, a baseline FPGA (Lattice iCE40UP5K) and eFPGA technology is used in our to be conducted experiments.
In IMDs with wireless connectivity, securing its communication is of utmost importance, especially in life-critical applications. Most commercial security implementations run on MCUs in software, which makes all but the most basic primitives infeasible to run. Hardware accelerated blocks within MCUs have been used for a tremendous performance and energy gain, however, no fast alternative is available in the case the accelerated primitive is broken during the lifetime of an implanted IMD. FPGAs could combine the advantages of both CPUs and ASIC peripherals and as seen from literature, cryptographic schemes are excellent candidates for low-power FPGA implementation, resource and energy wise. As such, hardware-accelerated cryptography in IMDs is considered as first use case in this thesis.
No IMD-suitable hardware-accelerated neural network could be found that fulfills all of our four requirements in literature. Designing an efficient ANN on a low-power FPGA is a difficult task, judging from the lack of well-documented suitable implementations. Looking at mapping engines to generate ANNs that do fulfill all requirements, time was too limiting to be able to use one for our experiments. The best readily available ANN that has a well-documented FPGA implementation is the Lattice CNN Compact Accelerator IP which, integrated in the Key Phrase Detector Project, will be taken as reference ANN for the second use case in our experiments.
Wireless energy harvesting was discussed with the goal to mitigate battery drain by adversaries. The maximum available continuous power draw was found to be 6.15mW from literature for the security primitives in Case 1 (Section 1.3.1), where the power limit of Case 2 (Section 1.3.2) will not be defined by this limit but rather indirectly by prospected battery life calculations.

In Chapter 3, our benchmark suite consisting of four algorithms, two interfaces and three groups of platforms is presented and designed. First, the implementations of the three selected security primitives (AES-128, SIMON-64/128 and PHOTON-128) are demonstrated. For every primitive, a premade IP core was selected that is optimized for low resource usage. With AES and SIMON being both block ciphers, interfacing them is done by presenting block and key data in parallel. Reading data from and writing to the PHOTON block is done in a serial fashion, as is fitting for hashing protocols which have a varying input message size. As neural network representative the Key Phrase Detector example project from Lattice is chosen, which is the last of our algorithm selection. The actual interfaces were created in Section 3.3, for simulating realistic resource overhead on the security primitives implemen-

tations. Two different scenarios were considered, as seen from the FPGA: off-die communication with an external host MCU and on-die communication in the case of an eFPGA integrated within an MCU on the same die. For both scenarios, fitting hardware implementations have been designed in the form of UART and AHB-lite interfaces. Where the UART interface is based on a simple 8-bit UART IP core, the AHB-Lite interface has been designed completely from scratch, with both interfaces requiring adaptations depending on the respective cryptographic IP core.
In Section 3.4, a set of two MCUs and a baseline FPGA have been selected to perform performance and energy comparisons for our experiments. Additionally, eFPGA technology will be considered. As is clear from the available ultra-low-power FPGAs currently on the market, the iCE40 UltraPlus series is the only affordable option. The MCU selection is based upon processors common in modern IMDs, with a higher end and energy-efficient version of the EFM32 Gecko series. This set of platforms is representable hardware for comparing a hypothetical FPGA-equipped IMD with a conventional MCU-based unit.
Next, eFPGAs are added to our platforms comparison in addition to MCU and FPGA hardware. An extra step is introduced in the design flow compared to regular FPGAs: that of designing the eFPGA fabric itself. With the Menta architecture being LUT6-based, the resource usage is different from our algorithms implemented on the baseline FPGA. Determining resource usage of the Key Phrase Detector on Menta fabric can only be calculated, as its HDL code is proprietary and non-portable. With an estimation based on linear trends between Lattice and Menta resource usage of all security primitives, the prospected LUT6 resource usage factor for the Key Phrase Detector is derived as being 75.33% of the LUT4 count on the Lattice FPGA. Having obtained this factor, eFPGA architectures that can just fit this CNN project were designed, as well as one fitting all security primitives, totaling three Menta architectures to be used in our experiments.
Last, our benchmark suite for current measurements on the iCE40UP5K-B-EVN platform is prepared. This involves modifying our security primitive HDL code to let them run at continuous load, that is, encryption. For the Key Phrase Detector project, no modifications were necessary as this design is already built to operate continuously.
MCU current measurements are to be performed by taking active current, which is identical for all implementations as its hardware is unchanged. Only measurements of the security primitives can be done, as no Key Phrase Detector software version is available.
Physical eFPGA measurements are not possible as the architectures only exist in simulation. However, static power can be obtained from Origami Designer and dynamic power from an experimental calculation model. As a global average activity factor needs to be provided in this model, multiple realistic values are assumed.

In Chapter 4, our results answering the four research question and problem statement are discussed. Based on resource usage, only one data point is taken per security primitive, as the UART and AHB-Lite interfaces have a similar resource footprint. The largest resource usage impact is made by the algorithms: choosing SIMON instead of AES as block cipher makes up for a factor 3.5 decrease in resources used. Due to the Menta architecture being LUT6 based rather than Lattice's LUT4, calculations were done to extract the supposed resource usage of the Key Phrase project on Menta fabric, which was found to be 75% that of Lattice. Seeing that all DSPs are used in the Key Phrase Detector implementation, it may be DSP limited, which can be alleviated in the Menta architectures by placing more DSPs. This could potentially result in less resource usage than calculated here. With this figure, the size of eFPGA Architecture 2 and 3 was determined.
Compared to the AES ASIC present in our MCUs, the FPGA implementation is 17% to 40% slower, but FPGAs can be used for retaining speedup over a wider variety of algorithms. Block ciphers in software are tolerable but hashing with PHOTON-128 is not, taking 81ms to hash 128 bits with the Tiny Gecko. The FPGA will complete this task in a mere 1073 cycles, or less than 0.1ms. Therefore, FPGAs are beneficial in IMDs considering performance while retaining functional flexibility.
But, the FPGA consumes 4.4x the energy compared to our AES ASIC, although this can be mitigated by employing a custom, more lightweight cipher. Replacing AES with SIMON results in an energy consumption reduction of more than 11 times. Next to cryptography, FPGA hardware accelerated neural networks are possible with only twice the energy consumption as AES under the same runtime.
Adding an FPGA to our baseline Tiny Gecko MCU results in a 63% drop from 3.76 to 1.40 years, rendering the addition unfeasible. However, with static FPGA current being the major contributor to average

power consumption, power gating provides a solution. We simulate our iCE40UP5K FPGA having a power efficient sleep mode, like Flash*Freeze from Microsemi IGLOO devices, as well as turning the FPGA off completely when idle. These scenarios account for a jump from 1.40 to 2.47 and 3.48 years respectively, resulting in the feasibility of FPGAs in IMDs considering battery life, with a decrease of only 7.5% in the last scenario.
Considering physical area, using TSMC 7nm the Menta eFPGAs are only 12% and 21% the area of the iCE40UP5K for cryptography-only and neural network support, respectively. Using a 180nm process results in an area of 20 to 100 times that of the Lattice FPGA, which is unfeasible. The selected technology for all further experiments is TSMC 28nm, which results in 40% to 70% of the Lattice FPGA area, making the eFPGA more area-efficient than the baseline FPGA. Only chip and fabric area is considered without surrounding components and interconnect in the case of the baseline FPGA, which would skew the area advantage even more in the direction of the eFPGAs. The eFPGA area is more dependent on process technology size than its architecture. Comparison to the iCE40UP5K FPGA is mainly a comparison of process technologies and can possibly be extended by comparing the FPGA + MCU to an eFPGA-equipped MCU in area. However, due to unavailability of the latter, this comparison was not made.
The battery life expectations of FPGA and eFPGA scenarios show that even with a very high modeled average activity factor of 30%, the eFPGAs are more efficient than the iCE40UP5K FPGA. Due to their low power consumption, operating the security primitives on harvested power is possible, with no recharging needed. When recharging every week is available, continuous neural network usage is feasible as expected battery life is 10 to 12 days. Otherwise, only up to 6.5 minutes active usage per day for either architecture is the maximum to retain energy feasibility.
To determine if daily FPGA reconfiguration is feasible in IMDs, a scenario was set up where an FPGA is configured to a cryptographic engine and operates for two minutes, after which the FPGA is configured back to its original state. Two parameters determine the feasibility: execution time and energy consumption added with a daily double reconfiguration. Regarding energy, it was found that the FPGA is faster than software AES, SIMON and PHOTON starting with messages of 8kiB, 16kiB antd 256 bits respectively, with an added 70.64ms configuration penalty on the FPGA side. The feasibility case was even stronger considering energy: software AES was at messages from 1kiB already less efficient than the FPGA with an added configuration cost of 28.82uJ. Within our use case, these message size thresholds will easily be reached. This renders FPGAs feasible in IMDs with both performance and energy configuration costs added.
No direct mention is made of FPGAs in European regulation, but multiple sections state regulations for software, which is therefore fully covered. Consulting certification experts, it was found that integrated circuits are generally considered as black boxes in regulation, meaning that only its functionality needs to be verified, but that it does not matter if an MCU or FPGA is inside the box. If verification of FPGAs can be brute forced by exhaustive testing of all possible input and output combinations, it is a hardware device falling under IEC 60601-1. Otherwise, IEC 62304 needs to be applied, which is the same regulation as for software components. This distinction is universally held by regulation companies and experts in certification. Therefore, regulation will be the same as with state-of-the-art MCU-based IMDs with no additional legal difficulty in place.

5.2. Main Contributions

This thesis centers around answering our problem statement, which is repeated here for convenience:

Can FPGAs be used as a reconfigurable fabric within IMDs?

Our contributions are grouped in answers on the four formulated research questions:

Can FPGA fabric be used in IMDs in terms of energy and execution time?

- We show that FPGA fabric is feasible and certainly beneficial if the active part of its duty cycle is not higher than a couple minutes per day and when the fabric is turned off or put into deep sleep when idle. With static power rising with smaller nodes such as those used in our eFPGA experiments, smart power gating when idle mitigates all these static power disadvantages. Energy feasibility

is therefore achieved.

- As a novelty, we considered hardware-accelerated artificial neural networks suitable for IMDs in this thesis. An extensive survey shows that no plug-and-play ANN is available that suits our needs and has a reference software implementation. However, feasibility for this use case is shown by seeing that it consumes about twice the energy as AES-128, while requiring 40% more FPGA resources. Having battery recharge capability at hand, hardware-accelerated neural networks in IMDs are within reach.

How big are the improvements of an eFPGA over an FPGA with regard to energy consumption and area?

- We designed three IMD-suited eFPGA architectures for comparison with an off-the-shelf FPGA in terms of area and energy feasibility in IMDs. We show that eFPGAs provide an advantage over regular FPGAs in both fields, occupying up to 21% less area and resulting in a 1.89 times longer battery life in the neural network case with two minutes of daily usage. Improvements over regular FPGAs are therefore significant and make eFPGAs worth considering for IMD manufacturers.
- We provide insight in the influence of different silicon process nodes on eFPGA + IMD feasibility. Old 180nm processes yield an area that is too large for eFPGA usage, whereas modern 28nm and 7nm processes provide noticeable area gains.

In what cases is daily FPGA reconfiguration beneficial in IMDs?

- We show that daily reconfiguration to a cryptographic peripheral in IMD FPGAs is feasible, if the peripheral is used on data sizes of at least 8kiB and 16kiB for AES and SIMON respectively. Hashing with PHOTON is already feasible in this case when operating on two blocks or 256 bits or more.

Having an FPGA-equipped IMD, what are the difficulties in aspect to legal certification?

- Unlike previous research, legal certification is considered as an additional feasibility factor. We show that FPGAs are treated as software devices, which have existing certification procedures in place. Therefore, no additional paperwork is required with an FPGA-equipped IMD compared to a MCU-based one that runs software. Answering the fourth research question: no additional difficulties are present.

Work that did not directly answer the problem statement is listed here:

- We created UART & AHB-Lite communication interfaces to pair with our selection of crypto IP cores for a realistic evaluation of the first use case. We show that the most deciding factor for energy and resource feasibility is not the communication interface, but the selected crypto IP core.

Seeing that we have answered all four research questions, the problem statement can be answered as such: FPGAs can certainly be used as reconfigurable fabric in IMDs under the conditions listed in our answers above. Before this research was conducted, it was not clear in what ways FPGAs would be manageable, primarily energy-wise, while preserving the typical multi-year battery life of IMDs. Having done the research, many obstacles of the technical and legal kind have been removed that hold IMD manufacturers from using FPGAs in their devices. As a result, an IMD manufacturer could implement an FPGA or eFPGA in their device, taking our feasibility conditions as a guideline without having to find out these conditions themselves.

5.3. Future work

- Find or design a tiny ANN that has both a hardware and software implementation available. In our research, it is clear that ANNs on tiny FPGAs are feasible for IMDs, but no MCU software reference point is included. Not only energy analysis could be conducted like done in this thesis, but a performance comparison would give indication at how much existing medical therapies can be improved. Also, an analysis of different existing ANN-friendly algorithms would give a clear picture on the algorithmic complexity that can be increased when using FPGAs instead of MCUs. Including this data point into the evaluation would give novel information and ground on why IMD ANNs need to be run on FPGAs.

- Analyze a neural network implementation that is actually meant for seizure detection or prediction or another complex medical therapy to gain a more precise insight in the feasibility of ANN-featuring medical therapies. A comparison with existing therapies can be performed to find out what therapies benefit from an FPGA implementation and which ones have an absolute need for hardware acceleration.
- Include dynamic power calculations for eFPGAs, deriving the average activity factor from actual testbench data instead of a global prediction. Using Synopsys Power Compiler for example could give the desired activity factor already. Additionally, replacing the used dynamic power prediction model with a thorough analysis of a hardened eFPGA design will result in power values that are more close to reality.
- Consider latency and data throughput performance of different communication interfaces, instead of only the resource usage as done in this thesis. Having more performance figures on this part can make a stronger case for using eFPGAs instead of FPGAs with applications that are heavy on data throughput. Performance comparisons between UART and AHB-Lite will lead to the conclusion that the communication overhead of the latter is vastly lower than UART with its economic interconnect. For communication-heavy implementations, eFPGAs could have a serious added performance advantage.
- Extend our FPGA vs eFPGA area comparison by taking into account actual PCB space, including surrounding components and interconnect in the case of a traditional separate FPGA chip. With this addition, the area benefit of using an eFPGA instead of an FPGA will certainly be even larger than it is in our analysis.
- add a fourth eFPGA architecture of around 1000 LUTs that can just hold SIMON or PHOTON-sized implementations. If the intended purpose of the FPGA fabric in IMDs is already very specific, for example only lightweight block ciphers, a much smaller eFPGA could suffice. With less LUTs, both static and dynamic power will be reduced, making it even more attractive to IMD manufacturers to incorporate an eFPGA in their devices.
- Implement an actual eFPGA, choosing one or more process technologies and perform IP hardening of the designed eFPGA architectures. With this addition, a more accurate picture could be presented on power, area, throughput and latency characteristics and optimization specifically for the ultra-low-power environment could be performed, whereas now only the most power hungry technology (TSMC 28nm) has been coarsely modeled.

References

[1] O. Aquilina. “A brief history of cardiac pacing.” In: (2006). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3232561.

[2] *ABBOTT’s Proclaim XR Spinal Cord Stimulator Gets FDA Approval for Chronic Pain Management*. https://www.wearable-technologies.com/2019/10/abbotts-proclaim-xr-spinal-cord-stimulator-gets-fda-approval-for-chronic-pain-management/. Accessed: 23-04-2020.

[3] Erwin Fuhrer. *Various neural implants and the location where they interface the central nervous system. This schematic display shows the wide spread DBS device and the compact CI system with their individual components on the right side and the location where they are implanted.* URL: https://www.researchgate.net/figure/Various-neural-implants-and-the-location-where-they-interface-the-central-nervous-system_fig1_323622113 (visited on 06/21/2021).

[4] M. M. Mansour et al. “Experimental investigation of wireless energy harvesting with a Bluetooth low energy sensing unit”. In: *2018 International Conference on Electronics Packaging and iMAPS All Asia Conference (ICEP-IAAC)*. 2018, pp. 189–193.

[5] Chunxiao Li, A. Raghunathan, and N. K. Jha. “Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system”. In: *2011 IEEE 13th International Conference on e-Health Networking, Applications and Services*. 2011, pp. 150–156.

[6] Matheus E. Garbelini, Sudipta Chattopadhyay, and Chundong Wang. “SweynTooth: Unleashing Mayhem over Bluetooth Low Energy”. In: (2020). URL: https://asset-group.github.io/disclosures/sweyntooth/.

[7] H. Kassiri et al. “Closed-Loop Neurostimulators: A Survey and A Seizure-Predicting Design Example for Intractable Epilepsy Treatment”. In: *IEEE Transactions on Biomedical Circuits and Systems* 11.5 (2017), pp. 1026–1040.

[8] M. Roukhami et al. “Very Low Power Neural Network FPGA Accelerators for Tag-Less Remote Person Identification Using Capacitive Sensors”. In: *IEEE Access* 7 (2019), pp. 102217–102231.

[9] Guy G. F. Lemieux et al. “TinBiNN: Tiny Binarized Neural Network Overlay in about 5, 000 4-LUTs and 5mW”. In: *CoRR* abs/1903.06630 (2019). arXiv: 1903.06630. URL: http://arxiv.org/abs/1903.06630.

[10] Inc. Lattice Semiconductor. *iCE40 UltraPlus - ML/AI Low Power FPGA*. 2021. URL: http://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus (visited on 05/28/2021).

[11] A. Bogdanov et al. “PRESENT: An Ultra-Lightweight Block Cipher”. In: *Cryptographic Hardware and Embedded Systems - CHES 2007*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466. ISBN: 978-3-540-74735-2.

[12] Christophe Cannière, Orr Dunkelman, and Miroslav Knežević. “Katan and Ktantan —A Family of Small and Efficient Hardware-Oriented Block Ciphers”. In: Jan. 2009, pp. 272–288.

[13] Muhammad Ali Siddiqi, Christian Doerr, and Christos Strydis. *IMDfence: Architecting a Secure Protocol for Implantable Medical Devices*. 2020. eprint: 2002.09546 (cs.CR).

[14] Shyamnath Gollakota et al. “They Can Hear Your Heartbeats: Non-Invasive Security for Implantable Medical Devices”. In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM ’11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 2–13. ISBN: 9781450307970. DOI: 10.1145/2018436.2018438. URL: https://doi.org/10.1145/2018436.2018438.

[15] Q. Yang et al. “An on-chip security guard based on zero-power authentication for implantable medical devices”. In: *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2014, pp. 531–534.

[16] Muhammad Ali Siddiqi and Christos Strydis. “IMD security vs. energy: are we tilting at windmills?” In: *Proceedings of the 16th ACM International Conference on Computing Frontiers* (Apr. 2019). DOI: 10.1145/3310273.3323421. URL: http://dx.doi.org/10.1145/3310273.3323421.

[17] P. Yalla and J. Kaps. “Lightweight Cryptography for FPGAs”. In: *2009 International Conference on Reconfigurable Computing and FPGAs*. 2009, pp. 225–230.

[18] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael*. 1999.

[19] Ray Beaulieu et al. “The SIMON and SPECK Families of Lightweight Block Ciphers”. In: *IACR Cryptol. ePrint Arch.* 2013 (2013), p. 404.

[20] Jian Guo, Thomas Peyrin, and Axel Poschmann. “The PHOTON Family of Lightweight Hash Functions”. In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–239. ISBN: 978-3-642-22792-9.

[21] Maria Hügle et al. “Early Seizure Detection with an Energy-Efficient Convolutional Neural Network on an Implantable Microcontroller”. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. 2018, pp. 1–7. DOI: 10.1109/IJCNN.2018.8489493.

[22] Simon Heller et al. “Hardware Implementation of a Performance and Energy-optimized Convolutional Neural Network for Seizure Detection”. In: *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2018, pp. 2268–2271. DOI: 10.1109/EMBC.2018.8512735.

[23] *tinyML*. 2021. URL: https://www.tinyml.org/about/ (visited on 05/28/2021).

[24] Christos Strydis et al. “A System Architecture, Processor and Communication Protocol for Secure Implants”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10 (Dec. 2013). DOI: 10.1145/2555289.2555313.

[25] M. Faizollah et al. “Low-power, small-size, generic controller dedicated to implantable biomedical microsystems”. In: *2012 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2012, pp. 204–207.

[26] S. Martínez and J. P. Oliver. “A low power FPGA based control unit for an implantable neuromodulation circuit”. In: *2019 X Southern Conference on Programmable Logic (SPL)*. 2019, pp. 63–68.

[27] S. Yamaguchi et al. “Programmable wireless sensor node featuring low-power FPGA and microcontroller”. In: *2013 International Joint Conference on Awareness Science and Technology Ubi-Media Computing (iCAST 2013 UMEDIA 2013)*. 2013, pp. 596–601.

[28] V. Rosello, J. Portilla, and T. Riesgo. “Ultra low power FPGA-based architecture for Wake-up Radio in Wireless Sensor Networks”. In: *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*. 2011, pp. 3826–3831.

[29] H. Qi, O. Ayorinde, and B. H. Calhoun. “An ultra-low-power FPGA for IoT applications”. In: *2017 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2017, pp. 1–3.

[30] Fei Zhang, Mehdi Aghagolzadeh, and Karim Oweiss. “A Fully Implantable, Programmable and Multimodal Neuroprocessor for Wireless, Cortically Controlled Brain-Machine Interface Applications”. In: *Signal Process Syst. 2012;69(3):351–361* (). DOI: 10.1007/s11265-012-0670-x.

[31] T. Zhan et al. “A Resource-Optimized VLSI Implementation of a Patient-Specific Seizure Detection Algorithm on a Custom-Made 2.2 cm^2 Wireless Device for Ambulatory Epilepsy Diagnostics”. In: *IEEE Transactions on Biomedical Circuits and Systems* 13.6 (2019), pp. 1175–1185.

[32] Neha T. *Block cipher in Electronic Codebook Mode.* URL: https://binaryterms.com/block-cipher.html (visited on 06/21/2021).

[33] *KLEIN*. 2017. URL: https://github.com/tomirio619/Midori (visited on 06/09/2021).

[34] *LED*. 2017. URL: https://github.com/tomirio619/Midori (visited on 06/09/2021).

[35] *Midori: Python and hardware implementation of Midori 128*. 2017. URL: https://github.com/tomirio619/Midori (visited on 06/09/2021).

[36] Guido Bertoni et al. *The KECCAK SHA-3 submission*. 2011. URL: https://keccak.team/keccak.html (visited on 06/21/2021).

[37] Salah Albermany. *Novel Design of RADG Automata In CRNs - Scientific Figure on ResearchGate.* URL: https://www.researchgate.net/figure/Figure-1-1-Symmetric-and-Asymmetric-Cryptosystem-Scheme-Asymmetric-cryptography_fig1_326331494 (visited on 06/21/2021).

[38] Menta & Secure-IC. *Insights at Menta & Secure-IC's Partnership & First Results.* URL: https://www.menta-efpga.com/videos (visited on 07/08/2021).

[39] T. Xu, J. B. Wendt, and M. Potkonjak. "Matched Digital PUFs for Low Power Security in Implantable Medical Devices". In: *2014 IEEE International Conference on Healthcare Informatics*. 2014, pp. 33–38.

[40] Siskos Dimitrios. "A Co-Processor for a Secure Implantable Medical Device". MA thesis. Delft University of Technology, 2011.

[41] Turki Al-Somani and Hilal Houssain. "Implementation of GF(2m) Elliptic Curve cryptoprocessor on a Nano FPGA". In: (Jan. 2011).

[42] Anita Aghaie et al. "Impeccable Circuits". In: *IEEE Transactions on Computers* 69.3 (2020), pp. 361–376. DOI: 10.1109/TC.2019.2948617.

[43] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

[44] Analytics Vidhya. *Artificial Neural Network, Its inspiration and the Working Mechanism*. URL: https://www.analyticsvidhya.com/blog/2021/04/artificial-neural-network-its-inspiration-and-the-working-mechanism/ (visited on 06/21/2021).

[45] Glosser.ca. *Own work, Derivative of File:Artificial neural network.svg, CC BY-SA 3.0*. URL: https://commons.wikimedia.org/w/index.php?curid=24913461 (visited on 06/21/2021).

[46] Sumit Saha. *A CNN sequence to classify handwritten digits*. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visited on 06/21/2021).

[47] Michael Copeland. *Inference is where capabilities learned during deep learning training are put to work.* URL: https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/ (visited on 06/21/2021).

[48] Bing Liu et al. "An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution". In: *Electronics* 8 (Mar. 2019), p. 281. DOI: 10.3390/electronics8030281.

[49] Rijad Sarić et al. "FPGA-based real-time epileptic seizure classification using Artificial Neural Network". In: *Biomedical Signal Processing and Control* 62 (Sept. 2020), p. 102106. DOI: 10.1016/j.bspc.2020.102106.

[50] Shuo Wang et al. "C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs". In: *CoRR* abs/1803.06305 (2018). arXiv: 1803.06305. URL: http://arxiv.org/abs/1803.06305.

[51] Ali Jahanshahi. "TinyCNN: A Tiny Modular CNN Accelerator for Embedded FPGA". In: *CoRR* abs/1911.06777 (2019). arXiv: 1911.06777. URL: http://arxiv.org/abs/1911.06777.

[52] Paolo Meloni et al. "Curbing the roofline: a scalable and flexible architecture for CNNs on FPGA". In: May 2016, pp. 376–383. DOI: 10.1145/2903150.2911715.

[53] Yuteng Zhou, Shrutika Redkar, and Xinming Huang. "Deep learning binary neural network on an FPGA". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 281–284. DOI: 10.1109/MWSCAS.2017.8052915.

[54] Zbigniew Hajduk. "Reconfigurable FPGA implementation of neural networks". In: *Neurocomputing* 308 (May 2018). DOI: 10.1016/j.neucom.2018.04.077.

[55] FÉlix Moreno et al. "Reconfigurable Hardware Architecture of a Shape Recognition System Based on Specialized Tiny Neural Networks With Online Training". In: *IEEE Transactions on Industrial Electronics* 56.8 (2009), pp. 3253–3263. DOI: 10.1109/TIE.2009.2022076.

[56] Cheng Fu et al. "Towards Fast and Energy-Efficient Binarized Neural Network Inference on FPGA". In: *CoRR* abs/1810.02068 (2018). arXiv: 1810.02068. URL: http://arxiv.org/abs/1810.02068.

[57] Shuang Liang et al. “FP-BNN: Binarized neural network on FPGA”. In: *Neurocomputing* 275 (2018), pp. 1072–1086. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.09.046. URL: https://www.sciencedirect.com/science/article/pii/S0925231217315655.

[58] Inc. Lattice. *Lattice CNN Compact Accelerator IP - Implement Machine Learning Inferencing in mWs*. URL: https://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores04/compactcnn (visited on 05/30/2021).

[59] *Key Phrase Detection Using Compact CNN Accelerator IP - Reference*. 2019. URL: www.latticesemi.com/-/media/LatticeSemi/Documents/ReferenceDesigns/JM/FPGA-RD-02066-1-0-Key-Phrase-Detection-Using-Compact-CNN-Accelerator-IP.ashx?document_id=52764 (visited on 05/28/2021).

[60] Microsoft. *SeeDot: Compiler for Low-Precision Machine Learning*. URL: https://www.microsoft.com/en-us/research/project/seedot-compiler-for-low-precision-machine-learning/ (visited on 05/30/2021).

[61] Microsoft. *EdgeML: The Edge Machine Learning Library*. URL: https://github.com/Microsoft/EdgeML (visited on 05/30/2021).

[62] M Wielgosz and M Karwatowski. “Mapping Neural Networks to FPGA-Based IoT Devices for Ultra-Low Latency Processing”. In: (July 2019). DOI: 10.3390/s19132981.

[63] D. Halperin et al. “Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008, pp. 129–142.

[64] Muhammad Ali Siddiqi and Christos Strydis. “Towards Realistic Battery-DoS Protection of Implantable Medical Devices”. In: *CoRR* abs/1904.06893 (2019). arXiv: 1904.06893. URL: http://arxiv.org/abs/1904.06893.

[65] A. N. Abdulfattah et al. “Performance Analysis of MICS-Based RF Wireless Power Transfer System for Implantable Medical Devices”. In: *IEEE Access* 7 (2019), pp. 11775–11784.

[66] P. Li and R. Bashirullah. “A Wireless Power Interface for Rechargeable Battery Operated Medical Implants”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 54.10 (2007), pp. 912–916.

[67] M. Galizzi et al. “An inertial and environmental wireless platform with advanced energy harvesting capabilities”. In: *2015 IEEE International Symposium on Inertial Sensors and Systems (ISISS) Proceedings*. 2015, pp. 1–4.

[68] Javier Castillo Villar. *AES-128 Verilog core*. 2004. URL: https://opencores.org/projects/systemcaes (visited on 05/28/2021).

[69] Anita Aghaie et al. *SIMON-64/128 VHDL core*. 2019. URL: https://github.com/emsec/ImpeccableCircuits/tree/master/SIMON/1-SIMON64-128_no_protection (visited on 05/28/2021).

[70] Guido Bertoni, Joan Daemen, and Michaël Peeters. “Cryptographic sponge functions”. In: 2011.

[71] Homer Hsing. *SHA-3 (KECCAK) Verilog core*. 2013. URL: https://opencores.org/projects/sha3 (visited on 05/28/2021).

[72] *TensorFlow - An end-to-end open source machine learning platform*. 2021. URL: https://www.tensorflow.org/ (visited on 05/28/2021).

[73] *Caffe - Deep Learning Framework*. 2021. URL: https://caffe.berkeleyvision.org/ (visited on 05/28/2021).

[74] *Keras: the Python deep learning API*. 2021. URL: https://keras.io/ (visited on 05/28/2021).

[75] Inc. Lattice. *Lattice sensAI Stack*. URL: https://www.latticesemi.com/sensAI (visited on 06/21/2021).

[76] *AMBA Bus Protocols - AXI, AHB, APB - Understanding Architecture and References*. URL: http://verificationexcellence.in/amba-bus-architecture/ (visited on 05/28/2021).

[77] IngenieroLoco. *Diagram showing UART timing*. URL: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter#/media/File:UART_timing_diagram.svg (visited on 06/21/2021).

[78] Eric Bainville. *FPGA Simple UART*. 2013. URL: http://www.bealto.com/fpga-uart_io.html (visited on 05/28/2021).

[79] Neha T. *AMBA 3 AHB-Lite Protocol*. URL: http://eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf (visited on 06/21/2021).

[80] *EFM32 Tiny Gecko 11 32-bit MCU*. URL: https://www.silabs.com/mcu/32-bit/efm32-tiny-gecko-tg11 (visited on 05/28/2021).

[81] *EFM32 Giant Gecko Series 1 32-bit MCU*. URL: https://www.silabs.com/mcu/32-bit/efm32-giant-gecko-gg11 (visited on 05/28/2021).

[82] Muhammad Ali Siddiqi, Wouter A. Serdijn, and Christos Strydis. "Zero-Power Defense Done Right: Shielding IMDs from Battery-Depletion Attacks". In: *Journal of Signal Processing Systems* 93.4 (Apr. 2020), pp. 421–437. DOI: 10.1007/s11265-020-01530-5. URL: https://doi.org/10.1007%5C%2Fs11265-020-01530-5.

[83] Microsemi Corp. *ZL70103: Ultra-Low-Power Implantable Medical Transceiver*. 2015. URL: https://www.microsemi.com/product-directory/implantable-medical-transceivers/3915-zl70103 (visited on 06/17/2021).

[84] Microsemi Corp. *IGLOO nano Low Power Flash FPGAs with Flash*Freeze Technology*. 2015. URL: https://www.microsemi.com/document-portal/doc_download/130695-ds0110-igloo-nano-low-power-flash-fpgas-datasheet (visited on 06/17/2021).

[85] *Medtronic - Recharging Your Neurostimulator*. URL: https://www.medtronic.com/uk-en/patients/treatments-therapies/drug-pump-chronic-pain/living-with/neurostimulators-recharging-your-neurostimulator.html (visited on 05/28/2021).

[86] Inc. Lattice Semiconductor. *iCE40 Programming and Configuration - Technical note*. 2015. URL: https://www.latticesemi.com/view_document?document_id=46502 (visited on 06/17/2021).

[87] Inc. Lattice Semiconductor. *iCE40 UltraPlus Family Data Sheet*. 2020. URL: https://www.latticesemi.com/view_document?document_id=51968 (visited on 06/17/2021).

[88] European Union. "Regulation (EU) 2017/745 of the european parliament and of the council of 5 April 2017 on medical devices, amending Directive 2001/73/EC, Regulation (EC) No 178/2002 and Regulation (EC) No 1223/2009 and repealing Council Directives 90/385/EEC and 93/42/EEC". In: (2017).

[89] Monir El Azzouzi. *Classifying medical devices according to MDR2017/745*. URL: https://www.easymedicaldevice.com (visited on 06/21/2021).

[90] Blue Pearl Software. "RTL Development and Testing for Medical Devices". In: (2017). URL: www.bluepearlsoftware.com/files/MedicalDevicesPaperFinal.pdf.

[91] International Electrotechnical Commission. "Medical electrical equipment - Part 1: General requirements for basic safety and essential performance". In: (2015). URL: https://www.iso.org/standard/65529.html.

[92] International Electrotechnical Commission. "Medical Device software - Software Life Cycle Plocesses". In: (2006). URL: https://www.iso.org/standard/38421.html.

[93] Jeff Gable. *Does FPGA code count as software?* 2020. URL: https://jeffgable.com/daily/does-fpga-code-count-as-software/ (visited on 06/04/2021).

[94] P.J. Tanzillo. *Controlling Risk in Medical Devices*. 2009. URL: https://www.machinedesign.com/archive/article/21829447/controlling-risk-in-medical-devices (visited on 06/04/2021).

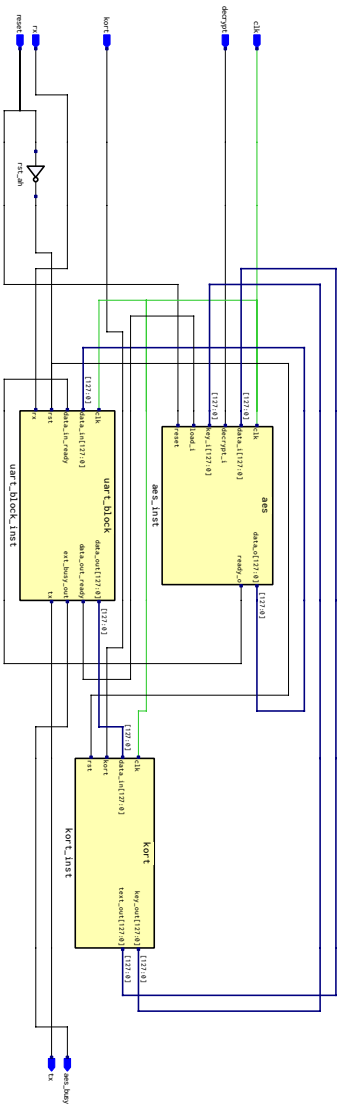
[95] Various. *Medical Device Software Development Lifecycle Standard Changes - ICE 62304:2006 vs. 62304:2015 (Amendment 1)*. 2016. URL: https://therealtimegroup.com/2016/07/10/medical-device-software-development-lifecycle-standard-changes-iec-623042006-vs-623042015-amendment-1/ (visited on 06/04/2021).

[96] Jomuna Choudhuri et al. "Frequently Asked Questions related to the Implementation of EN 62304:2006 with respect to MDD 93/42/EEC". In: (2013).

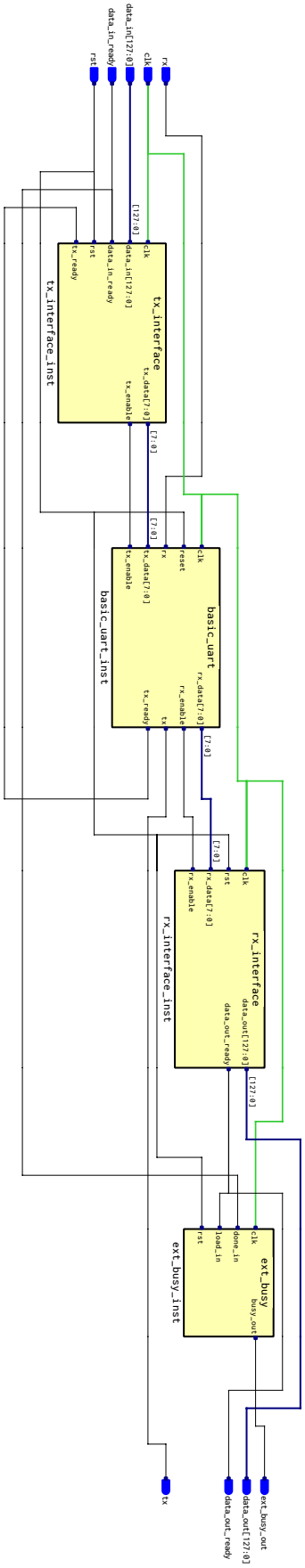
A

RTL schematics Crypto IP cores + UART

In this appendix, RTL schematics of the designed UART interfaces are depicted that are described in Chapter 3. The internals of the UART interfaces themselves are shown in Figures A.1b, A.2b and A.3b for our selected AES, SIMON and PHOTON IP cores respectively. The top-level IP core + interface RTL assemblies are found in Figures A.1a, A.2a and A.3a for AES, SIMON and PHOTON respectively. The AHB-Lite RTL schematics are not present in this appendix but can be found inline in Section 3.3.2.

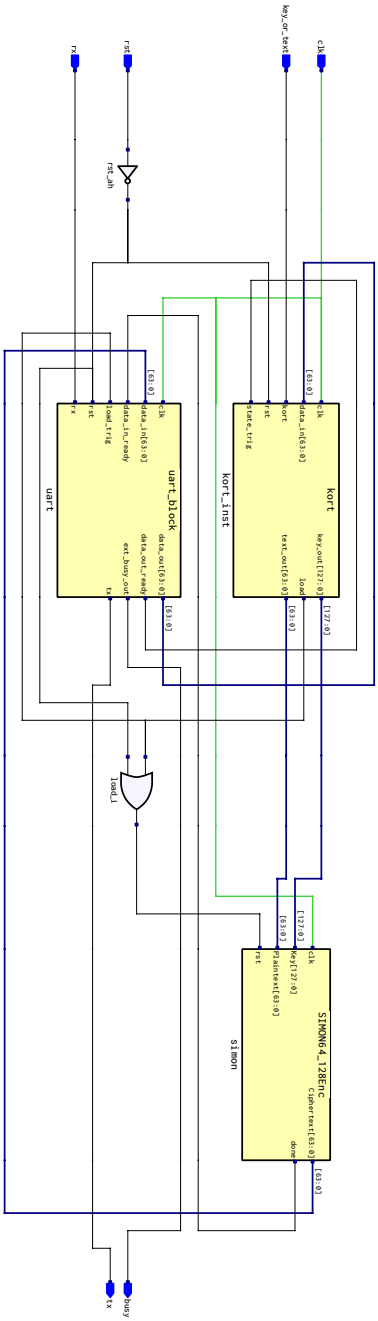


(a) AES + UART top-level

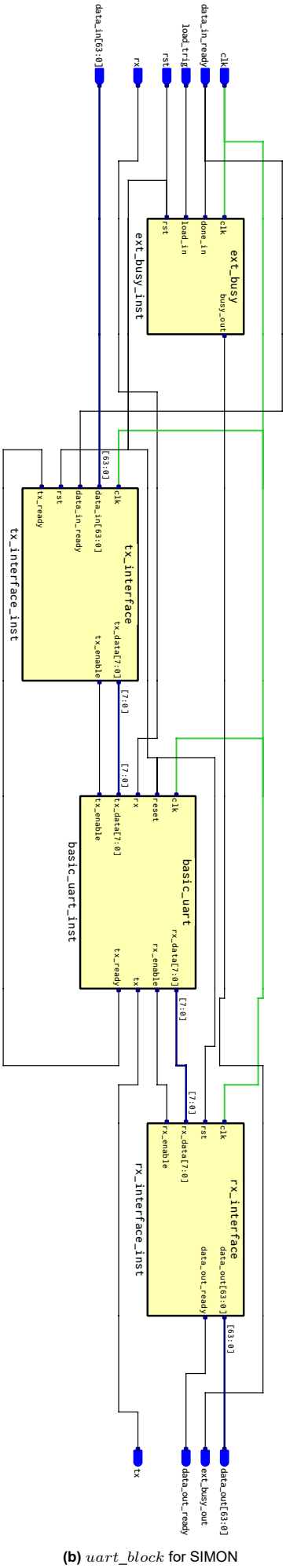


(b) $uart\_block$ for AES

Figure A.1: AES + UART assembly schematic

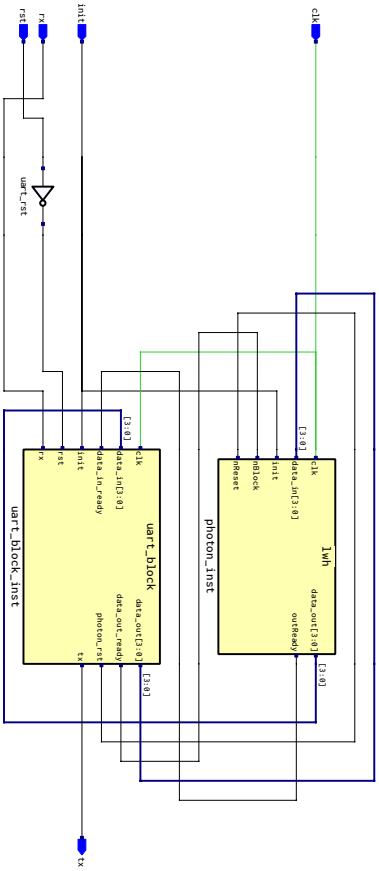


(a) SIMON + UART top-level

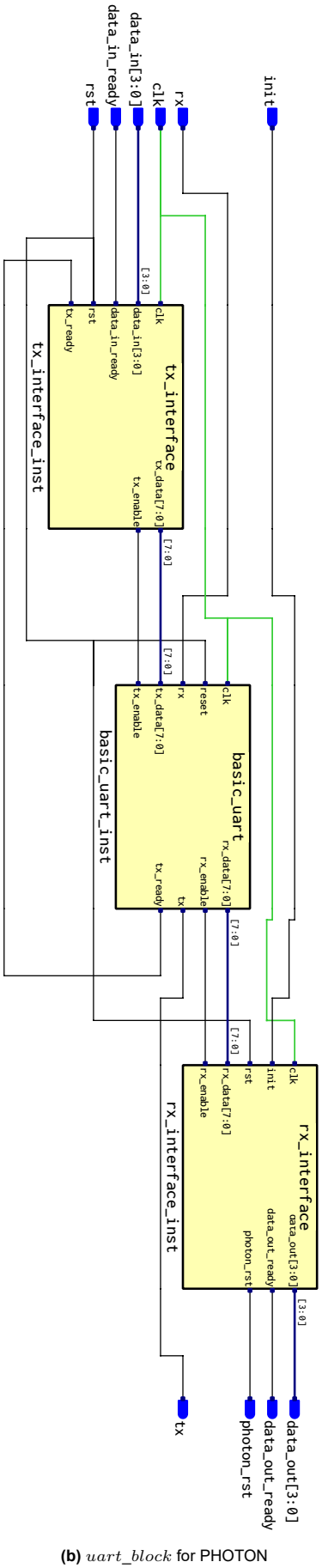


(b) $uart\_block$ for SIMON

Figure A.2: SIMON + UART assembly schematic



(a) PHOTON + UART top-level



(b) $uart\_block$ for PHOTON

Figure A.3: PHOTON + UART assembly schematic

B

Source code of UART and AHB-Lite interfaces

B.1. UART interfacing components

B.1.1. AES

tx_interface.v

```
1   module tx_interface (
2       input clk,
3       input rst,
4       input data_in_ready,
5       input tx_ready,
6       input [127:0] data_in,
7       output wire tx_data_ren,
8       output wire [7:0] tx_data,
9       output wire tx_enable
10      );
11      // ======================================================================
12      // Transmit side logic
13      // sends 16 bytes through the UART
14      // ======================================================================
15
16
17      // FSM state register
18      reg [4:0] trx_state_d;
19      reg [4:0] trx_state_q;
20      always @ (posedge clk) begin
21          if(rst)
22              trx_state_q <= 0;
23          else
24              trx_state_q <= trx_state_d;
25      end
26
27      // FSM state transition
28      always @ (trx_state_q or data_in_ready or data_in or tx_ready ) begin
29          case (trx_state_q)
30              0 : trx_state_d = (data_in_ready) ? 5'd17 : 5'd0; // go to extra lag
31                                                                // state first
32              1 : trx_state_d = (tx_ready) ? 5'd2  : 5'd1;
33              2 : trx_state_d = (tx_ready) ? 5'd3  : 5'd2;
34              3 : trx_state_d = (tx_ready) ? 5'd4  : 5'd3;
35              4 : trx_state_d = (tx_ready) ? 5'd5  : 5'd4;
36              5 : trx_state_d = (tx_ready) ? 5'd6  : 5'd5;
37              6 : trx_state_d = (tx_ready) ? 5'd7  : 5'd6;
38              7 : trx_state_d = (tx_ready) ? 5'd8  : 5'd7;
39              8 : trx_state_d = (tx_ready) ? 5'd9  : 5'd8;
40              9 : trx_state_d = (tx_ready) ? 5'd10 : 5'd9;
41              10: trx_state_d = (tx_ready) ? 5'd11 : 5'd10;
42              11: trx_state_d = (tx_ready) ? 5'd12 : 5'd11;
43              12: trx_state_d = (tx_ready) ? 5'd13 : 5'd12;
44              13: trx_state_d = (tx_ready) ? 5'd14 : 5'd13;
45              14: trx_state_d = (tx_ready) ? 5'd15 : 5'd14;
46              15: trx_state_d = (tx_ready) ? 5'd16 : 5'd15;
47              16: trx_state_d = (tx_ready) ? 5'd0  : 5'd16;
48
49              // extra lag state. Workaround for tx_ready coming one clk cycle too
50              // late when in state 0
51              17: trx_state_d = (tx_ready) ? 5'd1 : 5'd17;
52
53              default: trx_state_d = 5'd0;
54          endcase
55      end
56
57      // transmit register
58      reg [127:0] trx_reg_q;
59      wire [127:0] trx_reg_d;
60
61      always @ (posedge clk) begin
62          if(rst)
63              trx_reg_q <= 0;
64          else
65              trx_reg_q <= trx_reg_d;
66      end
```

```
67
68      assign trx_reg_d = (tx_data_ren) ? data_in : trx_reg_q;
69
70      assign tx_data_ren = (trx_state_q == 0 && !data_in_ready);
71
72      // LSB FIRST ENDIANNESS, same as at RX line
73      assign tx_data = (trx_state_q == 1)  ? trx_reg_q[7:0] :
74                       (trx_state_q == 2)  ? trx_reg_q[15:8] :
75                       (trx_state_q == 3)  ? trx_reg_q[23:16] :
76                       (trx_state_q == 4)  ? trx_reg_q[31:24] :
77                       (trx_state_q == 5)  ? trx_reg_q[39:32] :
78                       (trx_state_q == 6)  ? trx_reg_q[47:40] :
79                       (trx_state_q == 7)  ? trx_reg_q[55:48] :
80                       (trx_state_q == 8)  ? trx_reg_q[63:56] :
81                       (trx_state_q == 9)  ? trx_reg_q[71:64] :
82                       (trx_state_q == 10) ? trx_reg_q[79:72] :
83                       (trx_state_q == 11) ? trx_reg_q[87:80] :
84                       (trx_state_q == 12) ? trx_reg_q[95:88] :
85                       (trx_state_q == 13) ? trx_reg_q[103:96] :
86                       (trx_state_q == 14) ? trx_reg_q[111:104] :
87                       (trx_state_q == 15) ? trx_reg_q[119:112] :
88                       (trx_state_q == 16) ? trx_reg_q[127:120] :
89                       8'b0;
90
91
92      assign tx_enable = (!(trx_state_q == 0)) && tx_ready;
93  endmodule
```

rx_interface.v

```
1   module rx_interface (
2       input clk,
3       input rst,
4       input rx_enable,
5       input [7:0] rx_data,
6       input data_out_busy,
7       output wire [127:0] data_out,
8       output wire data_out_ready
9       );
10
11
12
13      // ============================================================================
14      // receive side logic
15      // waits for 16 bytes and then pushes them to data_out
16      // ============================================================================
17
18      // FSM States
19      //parameter RCV_WAIT_S = 3'd0;
20      //parameter RCV_1_ST = 3'd1;
21      //parameter RCV_2_ST = 3'd2;
22      //parameter RCV_3_ST = 3'd3;
23      //parameter RCV_4_ST = 3'd4;
24      //parameter RCV_5_ST = 3'd5;
25      //parameter RCV_PUSH_ST = 3'd6;
26
27      // FSM state register
28      reg [4:0] rcv_state_d;
29      reg [4:0] rcv_state_q;
30      always @ (posedge clk) begin
31          if(rst)
32              rcv_state_q <= 0;
33          else
34              rcv_state_q <= rcv_state_d;
35      end
36
37      // FSM state transition
38      always @ (rcv_state_q or rx_enable or rx_data or data_out_busy) begin
39          case (rcv_state_q)
40              0 : rcv_state_d = (rx_enable) ? 5'd1  : 5'd0;
41              1 : rcv_state_d = (rx_enable) ? 5'd2  : 5'd1;
42              2 : rcv_state_d = (rx_enable) ? 5'd3  : 5'd2;
43              3 : rcv_state_d = (rx_enable) ? 5'd4  : 5'd3;
44              4 : rcv_state_d = (rx_enable) ? 5'd5  : 5'd4;
45              5 : rcv_state_d = (rx_enable) ? 5'd6  : 5'd5;
46              6 : rcv_state_d = (rx_enable) ? 5'd7  : 5'd6;
47              7 : rcv_state_d = (rx_enable) ? 5'd8  : 5'd7;
48              8 : rcv_state_d = (rx_enable) ? 5'd9  : 5'd8;
49              9 : rcv_state_d = (rx_enable) ? 5'd10 : 5'd9;
50              10: rcv_state_d = (rx_enable) ? 5'd11 : 5'd10;
51              11: rcv_state_d = (rx_enable) ? 5'd12 : 5'd11;
52              12: rcv_state_d = (rx_enable) ? 5'd13 : 5'd12;
53              13: rcv_state_d = (rx_enable) ? 5'd14 : 5'd13;
54              14: rcv_state_d = (rx_enable) ? 5'd15 : 5'd14;
55              //added rx_enable to prevent race condition
56              15: rcv_state_d = (rx_enable && !data_out_busy) ? 5'd16 : 5'd15;
57              16: rcv_state_d = 5'd0; //data_out written in this state
58              default: rcv_state_d = 5'd0;
59          endcase
60      end
61
62      // receive (shift) register
63      wire rcv_reg_wen;
64
65      reg [7:0] rcv_reg_1_q;
66      reg [7:0] rcv_reg_2_q;
67      reg [7:0] rcv_reg_3_q;
68      reg [7:0] rcv_reg_4_q;
69      reg [7:0] rcv_reg_5_q;
70      reg [7:0] rcv_reg_6_q;
71      reg [7:0] rcv_reg_7_q;
72      reg [7:0] rcv_reg_8_q;
73      reg [7:0] rcv_reg_9_q;
74      reg [7:0] rcv_reg_10_q;
75      reg [7:0] rcv_reg_11_q;
76      reg [7:0] rcv_reg_12_q;
77      reg [7:0] rcv_reg_13_q;
78      reg [7:0] rcv_reg_14_q;
79      reg [7:0] rcv_reg_15_q;
80      reg [7:0] rcv_reg_16_q;
```

```
81
82      wire [7:0] rcv_reg_1_d;
83      wire [7:0] rcv_reg_2_d;
84      wire [7:0] rcv_reg_3_d;
85      wire [7:0] rcv_reg_4_d;
86      wire [7:0] rcv_reg_5_d;
87      wire [7:0] rcv_reg_6_d;
88      wire [7:0] rcv_reg_7_d;
89      wire [7:0] rcv_reg_8_d;
90      wire [7:0] rcv_reg_9_d;
91      wire [7:0] rcv_reg_10_d;
92      wire [7:0] rcv_reg_11_d;
93      wire [7:0] rcv_reg_12_d;
94      wire [7:0] rcv_reg_13_d;
95      wire [7:0] rcv_reg_14_d;
96      wire [7:0] rcv_reg_15_d;
97      wire [7:0] rcv_reg_16_d;
98
99      always @ (posedge clk) begin
100         if (rst) begin
101             rcv_reg_1_q <= 0;
102             rcv_reg_2_q <= 0;
103             rcv_reg_3_q <= 0;
104             rcv_reg_4_q <= 0;
105             rcv_reg_5_q <= 0;
106             rcv_reg_6_q <= 0;
107             rcv_reg_7_q <= 0;
108             rcv_reg_8_q <= 0;
109             rcv_reg_9_q <= 0;
110             rcv_reg_10_q <= 0;
111             rcv_reg_11_q <= 0;
112             rcv_reg_12_q <= 0;
113             rcv_reg_13_q <= 0;
114             rcv_reg_14_q <= 0;
115             rcv_reg_15_q <= 0;
116             rcv_reg_16_q <= 0;
117
118         end
119
120         else begin
121             rcv_reg_1_q <= rcv_reg_1_d;
122             rcv_reg_2_q <= rcv_reg_2_d;
123             rcv_reg_3_q <= rcv_reg_3_d;
124             rcv_reg_4_q <= rcv_reg_4_d;
125             rcv_reg_5_q <= rcv_reg_5_d;
126             rcv_reg_6_q <= rcv_reg_6_d;
127             rcv_reg_7_q <= rcv_reg_7_d;
128             rcv_reg_8_q <= rcv_reg_8_d;
129             rcv_reg_9_q <= rcv_reg_9_d;
130             rcv_reg_10_q <= rcv_reg_10_d;
131             rcv_reg_11_q <= rcv_reg_11_d;
132             rcv_reg_12_q <= rcv_reg_12_d;
133             rcv_reg_13_q <= rcv_reg_13_d;
134             rcv_reg_14_q <= rcv_reg_14_d;
135             rcv_reg_15_q <= rcv_reg_15_d;
136             rcv_reg_16_q <= rcv_reg_16_d;
137         end
138     end
139
140     assign rcv_reg_1_d = (rcv_reg_wen) ? rx_data : rcv_reg_1_q;
141     assign rcv_reg_2_d = (rcv_reg_wen) ? rcv_reg_1_q : rcv_reg_2_q;
142     assign rcv_reg_3_d = (rcv_reg_wen) ? rcv_reg_2_q : rcv_reg_3_q;
143     assign rcv_reg_4_d = (rcv_reg_wen) ? rcv_reg_3_q : rcv_reg_4_q;
144     assign rcv_reg_5_d = (rcv_reg_wen) ? rcv_reg_4_q : rcv_reg_5_q;
145     assign rcv_reg_6_d = (rcv_reg_wen) ? rcv_reg_5_q : rcv_reg_6_q;
146     assign rcv_reg_7_d = (rcv_reg_wen) ? rcv_reg_6_q : rcv_reg_7_q;
147     assign rcv_reg_8_d = (rcv_reg_wen) ? rcv_reg_7_q : rcv_reg_8_q;
148     assign rcv_reg_9_d = (rcv_reg_wen) ? rcv_reg_8_q : rcv_reg_9_q;
149     assign rcv_reg_10_d = (rcv_reg_wen) ? rcv_reg_9_q : rcv_reg_10_q;
150     assign rcv_reg_11_d = (rcv_reg_wen) ? rcv_reg_10_q : rcv_reg_11_q;
151     assign rcv_reg_12_d = (rcv_reg_wen) ? rcv_reg_11_q : rcv_reg_12_q;
152     assign rcv_reg_13_d = (rcv_reg_wen) ? rcv_reg_12_q : rcv_reg_13_q;
153     assign rcv_reg_14_d = (rcv_reg_wen) ? rcv_reg_13_q : rcv_reg_14_q;
154     assign rcv_reg_15_d = (rcv_reg_wen) ? rcv_reg_14_q : rcv_reg_15_q;
155     assign rcv_reg_16_d = (rcv_reg_wen) ? rcv_reg_15_q : rcv_reg_16_q;
156
157
158
159
160
161     // Output signals
162     assign data_out = {rcv_reg_1_q, rcv_reg_2_q, rcv_reg_3_q, rcv_reg_4_q,
163                        rcv_reg_5_q, rcv_reg_6_q, rcv_reg_7_q, rcv_reg_8_q,
164                        rcv_reg_9_q, rcv_reg_10_q, rcv_reg_11_q, rcv_reg_12_q,
165                        rcv_reg_13_q, rcv_reg_14_q, rcv_reg_15_q, rcv_reg_16_q};
166     assign data_out_ready = (rcv_state_q == 16) ? 1'b1 : 1'b0;
167     assign rcv_reg_wen = (!(rcv_state_q == 16)) && rx_enable;
168 endmodule
```

B.1.2. SIMON-64/128

tx_interface.v

```
1   module tx_interface (
2       input clk,
3       input rst,
4       input data_in_ready,
5       input tx_ready,
6       input [63:0] data_in,
7       //output wire tx_data_ren,
8       output wire [7:0] tx_data,
9       output wire tx_enable
10      );
11
12      //wire tx_data_ren;
13      // ======================================================================
14      // Transmit side logic
15      // sends 16 bytes through the UART
16      // ======================================================================
17
```

```
18
19      // FSM state register
20      reg [3:0] trx_state_d;
21      reg [3:0] trx_state_q;
22      always @ (posedge clk) begin
23          if(rst)
24              trx_state_q <= 0;
25          else
26              trx_state_q <= trx_state_d;
27      end
28
29      // FSM state transition
30      always @ (trx_state_q or data_in_ready or data_in or tx_ready ) begin
31          case (trx_state_q)
32              // go to extra lag state first
33              0 : trx_state_d = (data_in_ready) ? 4'd9 : 4'd0;
34              1 : trx_state_d = (tx_ready) ? 4'd2  : 4'd1;
35              2 : trx_state_d = (tx_ready) ? 4'd3  : 4'd2;
36              3 : trx_state_d = (tx_ready) ? 4'd4  : 4'd3;
37              4 : trx_state_d = (tx_ready) ? 4'd5  : 4'd4;
38              5 : trx_state_d = (tx_ready) ? 4'd6  : 4'd5;
39              6 : trx_state_d = (tx_ready) ? 4'd7  : 4'd6;
40              7 : trx_state_d = (tx_ready) ? 4'd8  : 4'd7;
41              8 : trx_state_d = (tx_ready) ? 4'd0  : 4'd8;
42
43              // extra lag state. Workaround for tx_ready coming one clk cycle too
44              // late when in state 0
45              9: trx_state_d = (tx_ready) ? 4'd1 : 4'd9;
46
47              default: trx_state_d = 4'd0;
48          endcase
49      end
50
51      // transmit register
52      reg [63:0] trx_reg_q;
53      wire [63:0] trx_reg_d;
54
55      reg tx_data_ren_q;
56      wire tx_data_ren_d;
57
58      always @ (posedge clk) begin
59          if(rst) begin
60              trx_reg_q <= 0;
61              tx_data_ren_q <= 0;
62          end
63          else begin
64              trx_reg_q <= trx_reg_d;
65              tx_data_ren_q <= tx_data_ren_d;
66          end
67      end
68
69      assign trx_reg_d = (tx_data_ren_q) ? data_in : trx_reg_q;
70
71      assign tx_data_ren_d = (trx_state_q == 0 && !data_in_ready);
72
73      // LSB FIRST ENDIANNESS, same as at RX line
74      assign tx_data = (trx_state_q == 1)  ? trx_reg_q[7:0] :
75                       (trx_state_q == 2)  ? trx_reg_q[15:8] :
76                       (trx_state_q == 3)  ? trx_reg_q[23:16] :
77                       (trx_state_q == 4)  ? trx_reg_q[31:24] :
78                       (trx_state_q == 5)  ? trx_reg_q[39:32] :
79                       (trx_state_q == 6)  ? trx_reg_q[47:40] :
80                       (trx_state_q == 7)  ? trx_reg_q[55:48] :
81                       (trx_state_q == 8)  ? trx_reg_q[63:56] :
82                       8'b0;
83
84
85      assign tx_enable = (!(trx_state_q == 0)) && tx_ready;
86  endmodule
```

rx_interface.v

```
1   module rx_interface (
2       input clk,
3       input rst,
4       input rx_enable,
5       input [7:0] rx_data,
6       output wire [63:0] data_out,
7       output wire data_out_ready//,
8       //input [1:0] rx_state
9       );
10
11      // ============================================================================
12      // receive side logic
13      // waits for 16 bytes and then pushes them to data_out
14      // ============================================================================
15
16      // FSM state register
17      reg [4:0] rcv_state_d;
18      reg [4:0] rcv_state_q;
19      always @ (posedge clk) begin
20          if(rst)
21              rcv_state_q <= 0;
22          else
23              rcv_state_q <= rcv_state_d;
24      end
25
26      // FSM state transition
27      always @ (rcv_state_q or rx_enable or rx_data) begin
28          case (rcv_state_q)
29              0 : rcv_state_d = (rx_enable) ? 4'd1 : 4'd0;
30              1 : rcv_state_d = (rx_enable) ? 4'd2 : 4'd1;
31              2 : rcv_state_d = (rx_enable) ? 4'd3 : 4'd2;
32              3 : rcv_state_d = (rx_enable) ? 4'd4 : 4'd3;
33              4 : rcv_state_d = (rx_enable) ? 4'd5 : 4'd4;
34              5 : rcv_state_d = (rx_enable) ? 4'd6 : 4'd5;
35              6 : rcv_state_d = (rx_enable) ? 4'd7 : 4'd6;
36              7 : rcv_state_d = (rx_enable) ? 4'd8 : 4'd7;
37              8 : rcv_state_d = 4'd0; //data_out written in this state
38              default: rcv_state_d = 5'd0;
```

```
        endcase
    end

    // receive (shift) register
    wire rcv_reg_wen;

    reg [7:0] rcv_reg_1_q;
    reg [7:0] rcv_reg_2_q;
    reg [7:0] rcv_reg_3_q;
    reg [7:0] rcv_reg_4_q;
    reg [7:0] rcv_reg_5_q;
    reg [7:0] rcv_reg_6_q;
    reg [7:0] rcv_reg_7_q;
    reg [7:0] rcv_reg_8_q;

    wire [7:0] rcv_reg_1_d;
    wire [7:0] rcv_reg_2_d;
    wire [7:0] rcv_reg_3_d;
    wire [7:0] rcv_reg_4_d;
    wire [7:0] rcv_reg_5_d;
    wire [7:0] rcv_reg_6_d;
    wire [7:0] rcv_reg_7_d;
    wire [7:0] rcv_reg_8_d;

    wire reg_clean = 0;

    always @ (posedge clk) begin
        if (rst) begin
            rcv_reg_1_q <= 0;
            rcv_reg_2_q <= 0;
            rcv_reg_3_q <= 0;
            rcv_reg_4_q <= 0;
            rcv_reg_5_q <= 0;
            rcv_reg_6_q <= 0;
            rcv_reg_7_q <= 0;
            rcv_reg_8_q <= 0;

        end

        else begin
            rcv_reg_1_q <= rcv_reg_1_d;
            rcv_reg_2_q <= rcv_reg_2_d;
            rcv_reg_3_q <= rcv_reg_3_d;
            rcv_reg_4_q <= rcv_reg_4_d;
            rcv_reg_5_q <= rcv_reg_5_d;
            rcv_reg_6_q <= rcv_reg_6_d;
            rcv_reg_7_q <= rcv_reg_7_d;
            rcv_reg_8_q <= rcv_reg_8_d;
        end
    end

    assign rcv_reg_1_d = reg_clean ? 0 : (rcv_reg_wen) ? rx_data : rcv_reg_1_q;
    assign rcv_reg_2_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_1_q : rcv_reg_2_q;
    assign rcv_reg_3_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_2_q : rcv_reg_3_q;
    assign rcv_reg_4_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_3_q : rcv_reg_4_q;
    assign rcv_reg_5_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_4_q : rcv_reg_5_q;
    assign rcv_reg_6_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_5_q : rcv_reg_6_q;
    assign rcv_reg_7_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_6_q : rcv_reg_7_q;
    assign rcv_reg_8_d = reg_clean ? 0 : (rcv_reg_wen) ? rcv_reg_7_q : rcv_reg_8_q;

    // Output signals
    assign data_out = {rcv_reg_1_q, rcv_reg_2_q, rcv_reg_3_q, rcv_reg_4_q,
                       rcv_reg_5_q, rcv_reg_6_q, rcv_reg_7_q, rcv_reg_8_q};
    assign data_out_ready = (rcv_state_q == 8) ? 1'b1 : 1'b0;

    assign rcv_reg_wen = (!(rcv_state_q == 8)) && rx_enable;
endmodule
```

B.1.3. PHOTON-128

tx_interface.v

```
// TX interface that stores incoming bytes from the PHOTON-128 block
// in a nibble shift register.
// This 4-bit wide shift register is chosen to specifically interface
// with the PHOTON-128 hash block, that has a 4-bit sequential output.
//
// Author : David Veselka
// Nov 2020



module tx_interface
#(
    parameter FSM_WAIT           = 0,
    parameter FSM_SHIFTIN        = 1,
    parameter FSM_SHIFTOUT_HIGH  = 2,
    parameter FSM_SHIFTOUT_LOW   = 3,
    parameter FSM_SHIFTOUT_WAIT  = 4,
    // The number of nibbles in the state matrix, 36 for PHOTON-128
    parameter NUM_NIBBLES        = 36
)
(
    input clk,
    input rst,
    input data_in_ready,
    input tx_ready,
    input [3:0] data_in,
    output wire [7:0] tx_data,
    output wire tx_enable
);

    //wire tx_data_ren;
    // ============================================================================
    // Transmit side logic
    // sends 18 bytes through the UART
    // ============================================================================
    integer i;

    reg [2:0] tx_state_d;
```

```
    reg [2:0] tx_state_q;
    reg [3:0] nibbleShiftReg_d[NUM_NIBBLES-1:0], nibbleShiftReg_q[NUM_NIBBLES-1:0];
    reg [7:0] tx_data_buf_d, tx_data_buf_q;
    reg       tx_enable_d, tx_enable_q;
    reg [6:0] nibble_cnt_d, nibble_cnt_q;

    // clk process
    always @ (posedge clk) begin
        if(rst) begin
            tx_state_q    <= 0;
            for(i = 0; i < NUM_NIBBLES; i = i + 1)
                nibbleShiftReg_q[i] <= 0;
            tx_data_buf_q <= 0;
            tx_enable_q   <= 0;
            nibble_cnt_q  <= 0;
        end
        else begin
            tx_state_q    <= tx_state_d;
            for(i = 0; i < NUM_NIBBLES; i = i + 1)
                nibbleShiftReg_q[i] <= nibbleShiftReg_d[i];
            tx_data_buf_q <= tx_data_buf_d;
            tx_enable_q   <= tx_enable_d;
            nibble_cnt_q  <= nibble_cnt_d;
        end
    end

    // comb process
    always @ (*) begin
        for(i = 0; i < NUM_NIBBLES; i = i + 1)
            nibbleShiftReg_d[i] <= nibbleShiftReg_q[i];
        tx_data_buf_d <= tx_data_buf_q;
        tx_enable_d   <= 1'b0;
        nibble_cnt_d  <= nibble_cnt_q;

        case(tx_state_q)
            FSM_WAIT:          begin
                if(data_in_ready) begin
                    // Clock first nibble & start shifting
                    tx_state_d <= FSM_SHIFTIN;
                    nibbleShiftReg_d[0] <= data_in;
                end
                else
                    tx_state_d <= FSM_WAIT;
            end
            FSM_SHIFTIN:       begin
                // Assuming data_in_ready is asserted for all nibbles and
                // deasserted immediately when all nibbles are clocked in
                if(data_in_ready) begin
                    tx_state_d <= FSM_SHIFTIN;
                    nibbleShiftReg_d[0] <= data_in;
                    for(i = 1; i < NUM_NIBBLES; i = i + 1)
                        nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
                end
                // at this point, all nibbles are in the shift
                // register and can be shifted out to the UART module
                else begin
                    tx_state_d <= FSM_SHIFTOUT_HIGH;
                end
            end
            // Shift one nibble to the high part of the output buffer
            FSM_SHIFTOUT_HIGH: begin
                tx_state_d <= FSM_SHIFTOUT_LOW;
                for(i = 1; i < NUM_NIBBLES; i = i + 1)
                    nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
                tx_data_buf_d[7:4] <= nibbleShiftReg_q[NUM_NIBBLES-1];
            end
            // Shift one nibble to the low part of the output buffer
            FSM_SHIFTOUT_LOW:  begin
                tx_state_d <= FSM_SHIFTOUT_WAIT;
                for(i = 1; i < NUM_NIBBLES; i = i + 1)
                    nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
                tx_data_buf_d[3:0] <= nibbleShiftReg_q[NUM_NIBBLES-1];
                nibble_cnt_d <= nibble_cnt_q + 1;
            end
            FSM_SHIFTOUT_WAIT: begin
                if(tx_ready) begin
                    if(nibble_cnt_q <= NUM_NIBBLES/2) begin
                        tx_state_d <= FSM_SHIFTOUT_HIGH;
                        tx_enable_d <= 1'b1;
                    end
                    else begin
                        tx_state_d <= FSM_WAIT;
                        nibble_cnt_d <= 0;
                    end
                end
                else
                    tx_state_d <= FSM_SHIFTOUT_WAIT;
            end
            default: begin
                tx_state_d <= FSM_WAIT;
            end
        endcase
    end

    assign tx_data = tx_data_buf_q;
    assign tx_enable = tx_enable_q;
endmodule
```

rx_interface.v

```
// RX interface that stores incoming UART bytes in a nibble shift register.
// This 4-bit wide shift register is chosen to specifically interface with the
// PHOTON-128 hash block, that requires a 4-bit sequential input.
//
// Author : David Veselka
// Oct 2020

module rx_interface
#(
    parameter FSM_WAIT         = 0,
```

```
11      parameter FSM_SHIFTIN_HIGH = 1,
12      parameter FSM_SHIFTIN_LOW  = 2,
13      parameter FSM_SHIFTOUT     = 3
14  )
15  (
16      input clk,
17      input rst,
18      input rx_enable,
19      input init,
20      input [7:0] rx_data,
21      output wire [3:0] data_out,
22      output wire data_out_ready,
23      output photon_rst
24  );
25
26      // ================================================================
27      // receive side logic
28      // ================================================================
29
30      reg[3:0] nibbleShiftReg_d[35:0], nibbleShiftReg_q[35:0];
31      reg[6:0] nibble_cnt_d, nibble_cnt_q;
32      reg[7:0] rx_data_buf_d, rx_data_buf_q;
33      //reg photon_rst_d, photon_rst_q;
34      integer i;
35
36      // FSM state register
37      reg [1:0] rcv_state_d;
38      reg [1:0] rcv_state_q;
39      always @ (posedge clk) begin
40          if(rst) begin
41              rcv_state_q <= 0;
42              nibble_cnt_q <= 0;
43              rx_data_buf_q <= 0;
44              //photon_rst_q <= 0;
45              for(i = 0; i < 36; i = i + 1)
46                  nibbleShiftReg_q[i] <= 4'hF;
47          end
48          else begin
49              rcv_state_q <= rcv_state_d;
50              nibble_cnt_q <= nibble_cnt_d;
51              rx_data_buf_q <= rx_data_buf_d;
52              //photon_rst_q <= photon_rst_d;
53              for(i = 0; i < 36; i = i + 1)
54                  nibbleShiftReg_q[i] <= nibbleShiftReg_d[i];
55          end
56      end
57
58      // FSM state transition
59      always @ (*) begin
60          // Equivalent of d <= q
61          rx_data_buf_d <= rx_data_buf_q;
62          nibble_cnt_d <= nibble_cnt_q;
63          //photon_rst_d <= photon_rst_q;
64          for(i = 0; i < 36; i = i + 1)
65              nibbleShiftReg_d[i] <= nibbleShiftReg_q[i];
66
67          case (rcv_state_q)
68                  FSM_WAIT:               begin
69                      if(nibble_cnt_q == 36) begin
70                          rcv_state_d <= FSM_SHIFTOUT;
71                          nibble_cnt_d <= 0;
72                      end
73                      else if(rx_enable) begin
74                          rcv_state_d <= FSM_SHIFTIN_HIGH;
75                          // Capture RX data for SHIFTIN states
76                          rx_data_buf_d <= rx_data;
77                      end
78                      else
79                          rcv_state_d <= FSM_WAIT;
80                  end
81                  FSM_SHIFTIN_HIGH:     begin
82                      rcv_state_d <= FSM_SHIFTIN_LOW;
83                      nibble_cnt_d <= nibble_cnt_q + 1;
84                      //shift in 2 nibbles from rx_data byte; high part first
85                      nibbleShiftReg_d[0] <= rx_data_buf_q[7:4];
86                      for(i = 1; i < 36; i = i + 1)
87                          nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
88                  end
89                  FSM_SHIFTIN_LOW:      begin
90                      rcv_state_d <= FSM_WAIT;
91                      nibble_cnt_d <= nibble_cnt_q + 1;
92                      //shift in 2 nibbles from rx_data byte; low part next
93                      nibbleShiftReg_d[0] <= rx_data_buf_q[3:0];
94                      for(i = 1; i < 36; i = i + 1)
95                          nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
96                  end
97                  FSM_SHIFTOUT:         begin
98                      //shift all nibbles out
99                      if(nibble_cnt_q == 35) begin
100                         rcv_state_d <= FSM_WAIT;
101                         nibble_cnt_d <= 0;
102                     end
103                     else begin
104                         nibble_cnt_d <= nibble_cnt_q + 1;
105                         for(i = 1; i < 36; i = i + 1)
106                             nibbleShiftReg_d[i] <= nibbleShiftReg_q[i-1];
107                         rcv_state_d <= FSM_SHIFTOUT;
108                     end
109                 end
110                 default:              begin
111                     rcv_state_d <= FSM_WAIT;
112                 end
113         endcase
114
115
116     end
117
118     assign data_out_ready = (rcv_state_q == FSM_SHIFTOUT) ? 1'b1 : 1'b0;
119     assign data_out = nibbleShiftReg_q[35];
120     assign photon_rst = data_out_ready || init;
121 endmodule
```

B.2. AHB interfacing components

B.2.1. AES-128

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////
// AHB interface for minimal AES-128 decrypt/encrypt block
// Author: David Veselka
// Sept/Oct 2020
//////////////////////////////////////////////////////////////

// Adresses 0xAA.... selects READ_STATUS
// Adresses 0xAB.... select  READ_TEXT
// Adresses 0xAC.... select  WRITE_KEY
// Adresses 0xAD.... select  DECRYPT states (WRITE_TEXT)
// Adresses 0xAE.... select  ENCRYPT states (WRITE_TEXT)
// Adresses 0xAF.... selects SENSE state

module ahb_interface
#(
    parameter FSM_SENSE             = 0,
    parameter FSM_WRITE_TEXT_2      = 1,
    parameter FSM_WRITE_TEXT_3      = 2,
    parameter FSM_WRITE_TEXT_4      = 3,
    parameter FSM_WRITE_KEY_2       = 4,
    parameter FSM_WRITE_KEY_3       = 5,
    parameter FSM_WRITE_KEY_4       = 6,
    parameter FSM_READ_TEXT_2       = 7,
    parameter FSM_READ_TEXT_3       = 8,
    parameter FSM_READ_TEXT_4       = 9,
    parameter FSM_INV_TRANS         = 10,
    parameter FSM_INV_RESP          = 11
)
(
    // Global signals
    input clk,
    input rst,

    // HREADY slave in omitted

    // Output
    output HREADY,
    output HRESP,
    output [31:0] HRDATA,

    // Select
    input HSEL,


    // Address and control
    input [31:0] HADDR,
    input HWRITE,
    input [2:0] HSIZE,
    input [1:0] HTRANS,
    input [2:0] HBURST,

    // Data
    input [31:0] HWDATA,

    // AES block interface signals
    input [127:0] aes_data_out,
    input aes_ready,

    output [127:0] aes_data_in,
    output [127:0] aes_key_in,
    output aes_decrypt_in,
    output aes_load
    );

    // ================================================
    // AES block registers
    reg [127:0] aes_text_d;
    reg [127:0] aes_key_d;
    reg aes_decrypt_d;
    reg aes_ready_d;
    reg [31:0] hrdata_d;

    reg [127:0] aes_text_q;
    reg [127:0] aes_key_q;
    reg aes_decrypt_q;
    reg aes_ready_q;
    reg [31:0] hrdata_q;

    // state register
    reg [3:0] state_d;
    reg [3:0] state_q;
    reg [3:0] state_q_prev;

    always @ (posedge clk) begin
        if(!rst) begin
            state_q <= FSM_INV_RESP;
            state_q_prev <= FSM_INV_RESP;
            aes_text_q <= 128'b0;
            aes_key_q <= 128'b0;
            aes_decrypt_q <= 1'b0;
            aes_ready_q <= 1'b0;
            hrdata_q <= 32'b0;
        end
        else begin
            state_q <= state_d;
            state_q_prev <= state_q;
            aes_text_q <= aes_text_d;
            aes_key_q <= aes_key_d;
            aes_decrypt_q <= aes_decrypt_d;
            aes_ready_q <= aes_ready_d;
            hrdata_q <= hrdata_d;
        end
    end

    // FSM toggle signals
```

```
107     wire addr_read_status  = HADDR[31:24] == 8'hAA;
108     wire addr_read_text    = HADDR[31:24] == 8'hAB;
109     wire addr_write_key    = HADDR[31:24] == 8'hAC;
110     wire addr_decrypt      = HADDR[31:24] == 8'hAD;
111     wire addr_encrypt      = HADDR[31:24] == 8'hAE;
112     wire addr_sense        = HADDR[31:24] == 8'hAF;
113
114     wire trans_idle       = HTRANS == 2'b00;
115     wire trans_busy       = HTRANS == 2'b01;
116     wire trans_nonseq     = HTRANS == 2'b10;
117     wire trans_seq        = HTRANS == 2'b11;
118
119     wire burst_single     = HBURST == 3'b000;
120     // No distinction between wrap and incr bursts needed
121     wire burst_quad       = HBURST == 3'b010 || HBURST == 3'b011;
122
123     wire [7:0] addr_offset;
124     wire addr_0 = addr_offset == 8'h00;
125     wire addr_4 = addr_offset == 8'h04;
126     wire addr_8 = addr_offset == 8'h08;
127     wire addr_b = addr_offset == 8'h0B;
128
129     //========== COMBINATORIAL BLOCK ==========//
130     always @ (*) begin
131
132         if(aes_ready)
133             aes_ready_d <= 1'b1;
134         else
135             aes_ready_d <= aes_ready_q;
136
137         aes_decrypt_d <= aes_decrypt_q;
138         aes_text_d    <= aes_text_q;
139         aes_key_d     <= aes_key_q;
140         hrdata_d      <= hrdata_q;
141
142         case (state_q)
143             FSM_SENSE: begin
144                 // Previously separate states, but data operations
145                 // are needed before the next clock cycle
146                 // FSM_DECRYPT & FSM_ENCRYPT
147                 if ((addr_decrypt ^ addr_encrypt) && HWRITE
148                     && burst_quad && trans_nonseq)   begin
149                     state_d <= FSM_WRITE_TEXT_2;
150                     aes_decrypt_d <= addr_decrypt;
151                     aes_text_d[127:96] <= HWDATA;
152                     aes_ready_d <= 1'b0;
153                 end
154                 // FSM_WRITE_KEY_1
155                 else if(addr_write_key   &&  HWRITE
156                     && burst_quad   && trans_nonseq && addr_0)   begin
157                     state_d <= FSM_WRITE_KEY_2;
158                     aes_key_d[127:96] <= HWDATA;
159                 end
160                 // FSM_READ_TEXT_1
161                 else if(addr_read_text   && !HWRITE
162                     && burst_quad   && trans_nonseq && addr_0)   begin
163                     state_d <= FSM_READ_TEXT_2;
164                     hrdata_d <= aes_data_out[127:96];
165                 end
166                 // FSM_READ_STATUS
167                 else if(addr_read_status && !HWRITE
168                     && burst_single && trans_nonseq && addr_0)   begin
169                     state_d <= FSM_SENSE;
170                     hrdata_d <= {30'b0, aes_decrypt_q, aes_ready_q};
171                 end
172                 else if(addr_sense) begin
173                     state_d <= FSM_SENSE;
174                 end
175                 else begin
176                     state_d <= FSM_INV_TRANS;
177                 end
178             end
179             //// WRITE TEXT (for decryption and encryption) ////
180             FSM_WRITE_TEXT_2: begin
181                 if(addr_4) begin
182                     state_d <= FSM_WRITE_TEXT_3;
183                     aes_text_d[95:64]  <= HWDATA;
184                 end
185                 else
186                     state_d <= FSM_INV_TRANS;
187             end
188             FSM_WRITE_TEXT_3: begin
189                 if(addr_8) begin
190                     state_d <= FSM_WRITE_TEXT_4;
191                     aes_text_d[63:32]  <= HWDATA;
192                 end
193                 else
194                     state_d <= FSM_INV_TRANS;
195             end
196             FSM_WRITE_TEXT_4: begin
197                 if((addr_decrypt || addr_encrypt) && burst_quad
198                     && trans_seq && addr_b) begin  // sanity check
199                     state_d <= FSM_SENSE;
200                     aes_text_d[31:0]   <= HWDATA;
201                 end
202                 else
203                     state_d <= FSM_INV_TRANS;
204             end
205             //// WRITE KEY ////
206             FSM_WRITE_KEY_2: begin
207                 if(addr_4) begin
208                     state_d <= FSM_WRITE_KEY_3;
209                     aes_key_d[95:64]  <= HWDATA;
210                 end
211                 else
212                     state_d <= FSM_INV_TRANS;
213             end
214             FSM_WRITE_KEY_3: begin
215                 if(addr_8) begin
216                     state_d <= FSM_WRITE_KEY_4;
217                     aes_key_d[63:32]  <= HWDATA;
218                 end
```

```
                else
                    state_d <= FSM_INV_TRANS;
            end
            FSM_WRITE_KEY_4: begin
                if(addr_write_key && HWRITE && burst_quad &&
                    trans_seq && addr_b)  begin // sanity check
                    state_d <= FSM_SENSE;
                    aes_key_d[31:0] <= HWDATA;
                end
                else
                    state_d <= FSM_INV_TRANS;
            end
            //// READ TEXT (for decryption and encryption) ////
            FSM_READ_TEXT_2:      begin
                if(addr_4) begin
                    state_d <= FSM_READ_TEXT_3;
                    hrdata_d <= aes_data_out[95:64];
                end
                else
                    state_d <= FSM_INV_TRANS;
            end
            FSM_READ_TEXT_3:      begin
                if(addr_8) begin
                    state_d <= FSM_READ_TEXT_4;
                    hrdata_d <= aes_data_out[63:32];
                end
                else
                    state_d <= FSM_INV_TRANS;
            end
            FSM_READ_TEXT_4:      begin
                if(addr_read_text && !HWRITE && burst_quad &&
                    trans_seq && addr_b) begin // sanity check
                    hrdata_d <= aes_data_out[31:0];
                    state_d <= FSM_SENSE;
                end
                else
                    state_d <= FSM_INV_TRANS;
            end
            //// INVALID ////
            FSM_INV_TRANS:              //invalid transaction
                state_d <= FSM_INV_RESP;
            FSM_INV_RESP:               //invalid response
                state_d <= FSM_SENSE;
            default:
                state_d <= FSM_SENSE;
        endcase
    end // end always for state transition

    assign addr_offset = HADDR[7:0];

    //========== OUTPUT ASSIGNMENTS ==========//

    // read data assignment
    assign HRDATA = hrdata_q;

    // Ready signal. No bus stall by HREADY = 0 implemented,
    // because reading data is always completed within one clock cycle
    assign HREADY = 1'b1;

    // Responses are always 1'b0 except during
    // two clock cycles around an invalid transaction
    assign HRESP = state_q == FSM_INV_TRANS
                   || state_q == FSM_INV_RESP ? 1'b1 : 1'b0;

    // AES block load control
    assign aes_load = (state_q_prev == FSM_WRITE_TEXT_4);
    assign aes_data_in = aes_text_q;
    assign aes_key_in = aes_key_q;
    assign aes_decrypt_in = aes_decrypt_q;
endmodule
```

B.2.2. SIMON-64/128

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////
// AHB interface for minimal SIMON-64/k128 decrypt/encrypt block
// Author: David Veselka
// Oct 2020
////////////////////////////////////////////////////////////

// Adresses 0xAA.... selects READ_STATUS
// Adresses 0xAB.... select  READ_TEXT
// Adresses 0xAC.... select  WRITE_KEY
// Adresses 0xAD.... select  DECRYPT states (WRITE_TEXT)
// Adresses 0xAE.... select  ENCRYPT states (WRITE_TEXT)
// Adresses 0xAF.... selects SENSE state

module ahb_interface
#(
    parameter FSM_SENSE             = 0,
    parameter FSM_WRITE_TEXT_2      = 1,
    //parameter FSM_WRITE_TEXT_3      = 2,
    //parameter FSM_WRITE_TEXT_4      = 3,
    parameter FSM_WRITE_KEY_2       = 4,
    parameter FSM_WRITE_KEY_3       = 5,
    parameter FSM_WRITE_KEY_4       = 6,
    parameter FSM_READ_TEXT_2       = 7,
    //parameter FSM_READ_TEXT_3       = 8,
    //parameter FSM_READ_TEXT_4       = 9,
    parameter FSM_INV_TRANS         = 10,
    parameter FSM_INV_RESP          = 11
)
(
    // Global signals
    input clk,
    input rst,

    // HREADY slave in omitted

    // Output
```

```
38      output HREADY,
39      output HRESP,
40      output [31:0] HRDATA,
41
42      // Select
43      input HSEL,
44
45
46      // Address and control
47      input [31:0] HADDR,
48      input HWRITE,
49      input [2:0] HSIZE,
50      input [1:0] HTRANS,
51      input [2:0] HBURST,
52
53      // Data
54      input [31:0] HWDATA,
55
56      // SIMON block interface signals
57      input [63:0] simon_data_out,
58      input simon_done,
59
60      output [63:0] simon_data_in,
61      output [127:0] simon_key_in,
62      output simon_load
63      );
64
65      // ================================================
66      // SIMON block registers
67      reg [63:0] simon_text_d;
68      reg [127:0] simon_key_d;
69      reg simon_decrypt_d;
70      reg simon_ready_d;
71      reg [31:0] hrdata_d;
72      reg [63:0] simon_data_out_d; // as seen from crypto block
73
74      reg [63:0] simon_text_q;
75      reg [127:0] simon_key_q;
76      reg simon_decrypt_q;
77      reg simon_ready_q;
78      reg [31:0] hrdata_q;
79      reg [63:0] simon_data_out_q; // as seen from crypto block
80
81
82      // state register
83      reg [3:0] state_d;
84      reg [3:0] state_q;
85      reg [3:0] state_q_prev;
86
87      always @ (posedge clk) begin
88          if(!rst) begin
89              state_q <= FSM_INV_RESP;
90              state_q_prev <= FSM_INV_RESP;
91              simon_text_q <= 64'b0;
92              simon_key_q <= 128'b0;
93              simon_decrypt_q <= 1'b0;
94              simon_ready_q <= 1'b0;
95              hrdata_q <= 32'b0;
96              simon_data_out_q <= 64'b0;
97          end
98          else begin
99              state_q <= state_d;
100             state_q_prev <= state_q;
101             simon_text_q <= simon_text_d;
102             simon_key_q <= simon_key_d;
103             simon_decrypt_q <= simon_decrypt_d;
104             simon_ready_q <= simon_ready_d;
105             hrdata_q <= hrdata_d;
106             simon_data_out_q <= simon_data_out_d;
107         end
108     end
109
110     // FSM toggle signals
111     wire addr_read_status  = HADDR[31:24] == 8'hAA;
112     wire addr_read_text    = HADDR[31:24] == 8'hAB;
113     wire addr_write_key    = HADDR[31:24] == 8'hAC;
114     wire addr_decrypt      = HADDR[31:24] == 8'hAD;
115     wire addr_encrypt      = HADDR[31:24] == 8'hAE;
116     wire addr_sense        = HADDR[31:24] == 8'hAF;
117
118     wire trans_idle       = HTRANS == 2'b00;
119     wire trans_busy       = HTRANS == 2'b01;
120     wire trans_nonseq     = HTRANS == 2'b10;
121     wire trans_seq        = HTRANS == 2'b11;
122
123     wire burst_single     = HBURST == 3'b000;
124     wire burst_undef      = HBURST == 3'b001;
125     // No distinction between wrap and incr bursts needed
126     wire burst_quad       = HBURST == 3'b010 || HBURST == 3'b011;
127
128     wire [7:0] addr_offset;
129     wire addr_0 = addr_offset == 8'h00;
130     wire addr_4 = addr_offset == 8'h04;
131     wire addr_8 = addr_offset == 8'h08;
132     wire addr_b = addr_offset == 8'h0B;
133
134     //========== COMBINATORIAL BLOCK ==========//
135     always @ (*) begin
136
137         if(simon_done) begin
138             simon_ready_d <= 1'b1;
139             simon_data_out_d <= simon_data_out;
140         end
141         else begin
142             simon_ready_d <= simon_ready_q;
143             simon_data_out_d <= simon_data_out_q;
144         end
145
146         simon_decrypt_d <= simon_decrypt_q;
147         simon_text_d    <= simon_text_q;
148         simon_key_d     <= simon_key_q;
149         hrdata_d      <= hrdata_q;
```

```
150
151         case (state_q)
152             FSM_SENSE: begin
153                 // Previously separate states, but data operations
154                 // are needed before the next clock cycle
155                 // FSM_DECRYPT & FSM_ENCRYPT
156                 if ((addr_decrypt ^ addr_encrypt) && HWRITE
157                     && burst_undef && trans_nonseq)   begin
158                     state_d <= FSM_WRITE_TEXT_2;
159                     //simon_decrypt_d <= addr_decrypt;
160                     simon_text_d[63:32] <= HWDATA;
161                     simon_ready_d <= 1'b0;
162                 end
163                 // FSM_WRITE_KEY_1
164                 else if(addr_write_key   &&  HWRITE &&
165                     burst_quad   && trans_nonseq && addr_0)   begin
166                     state_d <= FSM_WRITE_KEY_2;
167                     simon_key_d[127:96] <= HWDATA;
168                 end
169                 // FSM_READ_TEXT_1
170                 else if(addr_read_text   && !HWRITE &&
171                     burst_undef  && trans_nonseq && addr_0)   begin
172                     state_d <= FSM_READ_TEXT_2;
173                     hrdata_d <= simon_data_out_q[63:32];
174                 end
175                 // FSM_READ_STATUS
176                 else if(addr_read_status && !HWRITE &&
177                     burst_single && trans_nonseq && addr_0)   begin
178                     state_d <= FSM_SENSE;
179                     hrdata_d <= {30'b0, simon_decrypt_q, simon_ready_q};
180                 end
181                 else if(addr_sense) begin
182                     state_d <= FSM_SENSE;
183                 end
184                 else begin
185                     state_d <= FSM_INV_TRANS;
186                 end
187             end
188             //// WRITE TEXT (for decryption and encryption) ////
189             FSM_WRITE_TEXT_2: begin
190                 if(addr_4 && HWRITE && burst_undef && trans_seq
191                     && (addr_decrypt || addr_encrypt)) begin
192                     //state_d <= FSM_WRITE_TEXT_3;
193                     state_d <= FSM_SENSE;
194                     simon_text_d[32:0]  <= HWDATA;
195                 end
196                 else
197                     state_d <= FSM_INV_TRANS;
198             end
199             //// WRITE KEY ////
200             FSM_WRITE_KEY_2: begin
201                 if(addr_4) begin
202                     state_d <= FSM_WRITE_KEY_3;
203                     simon_key_d[95:64]  <= HWDATA;
204                 end
205                 else
206                     state_d <= FSM_INV_TRANS;
207             end
208             FSM_WRITE_KEY_3: begin
209                 if(addr_8) begin
210                     state_d <= FSM_WRITE_KEY_4;
211                     simon_key_d[63:32]  <= HWDATA;
212                 end
213                 else
214                     state_d <= FSM_INV_TRANS;
215             end
216             FSM_WRITE_KEY_4: begin
217                 if(addr_write_key && HWRITE && burst_quad &&
218                     trans_seq && addr_b)  begin  // sanity check
219                     state_d <= FSM_SENSE;
220                     simon_key_d[31:0] <= HWDATA;
221                 end
222                 else
223                     state_d <= FSM_INV_TRANS;
224             end
225             //// READ TEXT (for decryption and encryption) ////
226             FSM_READ_TEXT_2:      begin
227                 if(addr_4 && addr_read_text && !HWRITE && burst_undef
228                     && trans_seq) begin
229                     //state_d <= FSM_READ_TEXT_3;
230                     state_d <= FSM_SENSE;
231                     hrdata_d <= simon_data_out_q[31:0];
232                 end
233                 else
234                     state_d <= FSM_INV_TRANS;
235             end
236             //// INVALID ////
237             FSM_INV_TRANS:               //invalid transaction
238                 state_d <= FSM_INV_RESP;
239             FSM_INV_RESP:                //invalid response
240                 state_d <= FSM_SENSE;
241             default:
242                 state_d <= FSM_SENSE;
243         endcase
244     end // end always for state transition
245
246     assign addr_offset = HADDR[7:0];
247
248     //========== OUTPUT ASSIGNMENTS ==========//
249
250     // read data assignment
251     assign HRDATA = hrdata_q;
252
253     // Ready signal. No bus stall by HREADY = 0 implemented,
254     // because reading data is always completed within one clock cycle
255     assign HREADY = 1'b1;
256
257     // Responses are always 1'b0 except during two clock cycles
258     // around an invalid transaction
259     assign HRESP = state_q == FSM_INV_TRANS
260                    || state_q == FSM_INV_RESP ? 1'b1 : 1'b0;
261
```

```
    // AES block load control
    assign simon_load = (state_q_prev == FSM_WRITE_TEXT_2);
    assign simon_data_in = simon_text_q;
    assign simon_key_in = simon_key_q;
    //assign simon_decrypt_in = simon_decrypt_q;
endmodule
```

B.2.3. PHOTON-128

Initialization Interface

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Interface that takes care of initializing the external
// PHOTON block and directing input and output data to the host
// communication block.
//
// Originally meant to accept a 144-bit input (hence "parallel" interface) but
// reduced to a control unit accepting nibbles at its input from a shift register
// in the host communication block.
// Author: David Veselka
// Nov 2020
//////////////////////////////////////////////////////////////////////////////////

module lwh_par_interface
#(
    parameter FSM_WRITE     = 0,
    //parameter FSM_READ      = 1,
    parameter FSM_WAITRESET = 1,
    parameter FSM_SHIFTIN   = 2,
    parameter FSM_SHIFTOUT  = 3,
    parameter VEC_WIDTH     = 143,
    parameter NUM_NIBBLES   = 36
)
(
    input  clk,
    input  rst,
    input  init,                    // From AHB
    input  [3:0] lwh_nibble_in,     // From LWH
    input  [3:0] funnel_nibble_in,  // From AHB
    input  lwh_outReady,            // To LWH
    output lwh_init,                // To LWH (control)
    output lwh_nBlock,              // To LWH (control)
    output lwh_nReset,              // To LWH (control)
    output status,                  // To AHB. 0 = ready for read/write,
                                    // 1 = busy hashing
    output funnel_shift,            // To AHB
    output [3:0] lwh_nibble_out,    // To LWH
    output [3:0] funnel_nibble_out  // to AHB
    );

    reg [5:0] i_d, i_q;
    reg [2:0] state_d, state_q;


    // CLK process; active low synchronous reset
    always @ (posedge clk) begin
        if(!rst) begin
            state_q <= FSM_WRITE;
            i_q <= 0;
        end
        else begin
            state_q <= state_d;
            i_q <= i_d;
        end
    end

    // COMB process
    always @ (*) begin
        state_d <= state_q;
        i_d <= i_q;

        case(state_q)
            FSM_WRITE:    begin
                i_d <= 0;
                if(init) begin
                    state_d <= FSM_WAITRESET;
                end
            end
            // Extra state introduced to fully initialize the PHOTON block
            FSM_WAITRESET: begin
                if(i_q == 38) begin // 38 chosen empirically
                    state_d <= FSM_SHIFTIN;
                    i_d <= 0;
                end
                else i_d <= i_q + 1;
            end
            FSM_SHIFTIN:  begin
                if(i_q < NUM_NIBBLES-1) begin
                    i_d <= i_q + 1;
                end
                else begin
                    state_d <= FSM_SHIFTOUT;
                    i_d <= 0;
                end

            end
            FSM_SHIFTOUT : begin
                if(lwh_outReady) begin
                    if(i_q < NUM_NIBBLES-1) begin
                        i_d <= i_q + 1;
                    end
                    else begin
                        state_d <= FSM_WRITE;
                    end
                end
            end

            default: begin
                state_d <= FSM_WRITE;
```

```
            end
        endcase
    end

////////////// PHOTON control outputs ////////////////////////////////////////
// Data Out Ready (lwh_nBlock): only asserted while clocking IN              //
// the Initial Value nibbles.//                                              //
// NOT asserted when clocking OUT Hash nibbles.                              //
//                                                                           //
//                                                                           //
// PHOTON Init (lwh_init): asserted AFTER clocking IN the IV nibbles is done //
// and Data Out Ready is deasserted.                                         //
// ONLY deasserted at the rising edge of the last Hash OUT nibble.           //
//                                                                           //
//                                                                           //
// PHOTON Reset (lwh_nReset): asserted as long as                            //
// PHOTON Init OR Data Out Ready is asserted.                                //
///////////////////////////////////////////////////////////////////////////////



// Control & Status outputs
assign lwh_init   = state_q == FSM_SHIFTOUT ? 1 : 0;
assign lwh_nBlock = state_q == FSM_SHIFTIN  ? 1 : 0;
assign lwh_nReset = lwh_init || lwh_nBlock;
assign status = state_q == FSM_WRITE ? 1'b0 : state_q == FSM_SHIFTIN
           || state_q == FSM_SHIFTOUT ? 1'b1 : 1'b0;
assign funnel_shift = state_q == FSM_SHIFTIN
                  || (state_q == FSM_SHIFTOUT && lwh_outReady) ? 1 : 0;

// Data outputs
assign lwh_nibble_out = state_q == FSM_SHIFTIN ? funnel_nibble_in : 0;
assign funnel_nibble_out = state_q == FSM_SHIFTOUT
                      && lwh_outReady ? lwh_nibble_in : 4'b0;
endmodule
```

AHB-Lite interface

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////
// AHB interface for minimal PHOTON-128 decrypt/encrypt block
// Author: David Veselka
// Oct/Nov 2020
//
// TODO: re-introduce address offset checking (minimal LUTs!)
//////////////////////////////////////////////////////////////

// Adresses 0xAA.... selects READ_STATUS
// Adresses 0xAB.... select  HASHOUT
// Adresses 0xAC.... select  LOAD_IV
// Adresses 0xAF.... selects SENSE state

module ahb_interface
#(
    parameter FSM_SENSE         = 0,
    parameter FSM_LOAD_IV       = 1,
    parameter FSM_HASHOUT       = 2,
    parameter FSM_READ_STATUS   = 3,
    parameter FSM_INV_TRANS     = 4,
    parameter FSM_INV_RESP      = 5,
    parameter NUM_NIBBLES       = 36,
    parameter VEC_WIDTH         = 143
)
(
    // Global signals
    input clk,
    input rst,

    // AHB signals
    // HREADY slave in omitted
    output HREADY,
    output HRESP,
    output [31:0] HRDATA,
    input HSEL,
    input [31:0] HADDR,
    input HWRITE,
    input [2:0] HSIZE,
    input [1:0] HTRANS,
    input [2:0] HBURST,
    input [31:0] HWDATA,

    // PHOTON block interface signals
    input [3:0] funnel_nibble_in,
    input status,
    input funnel_shift,
    output photon_init,
    output [3:0] funnel_nibble_out
    );

    // ================================================
    // PHOTON block registers
    // The complete PHOTON state matrix in one vector.
    //Has a 4-bit input and output funnel at LSB and MSB side respectively.
    reg [143:0] photon_mat_d, photon_mat_q;
    reg [15:0] addr_buf_d, addr_buf_q;

    // Give the LWH a kick to start
    reg photon_init_q, photon_init_d;


    // state register
    reg [2:0] state_d, state_q;

    always @ (posedge clk) begin
        if(!rst) begin
            state_q <= FSM_INV_RESP;
            photon_mat_q <= 0;
            photon_init_q <= 0;
            addr_buf_q <= 0;
        end
```

```
73          else begin
74              state_q <= state_d;
75              photon_mat_q <= photon_mat_d;
76              photon_init_q <= photon_init_d;
77              addr_buf_q <= addr_buf_d;
78          end
79      end
80
81      // FSM toggle signals. Signals labeled *init are
82      // to trigger an FSM transition.
83      // Without init: delayed by one clock cycle to check
84      // validity in load and write states.
85      wire addr_read_status_init  = HADDR[31:24] == 8'hAA;
86      wire addr_hashout_init      = HADDR[31:24] == 8'hAB;
87      wire addr_load_iv_init      = HADDR[31:24] == 8'hAC;
88      wire addr_sense_init        = HADDR[31:24] == 8'hAF;
89
90      wire addr_read_status  = addr_buf_q[15:8] == 8'hAA;
91      wire addr_hashout      = addr_buf_q[15:8] == 8'hAB;
92      wire addr_load_iv      = addr_buf_q[15:8] == 8'hAC;
93      wire addr_sense        = addr_buf_q[15:8] == 8'hAF;
94
95      wire trans_idle       = HTRANS == 2'b00;
96      //wire trans_busy       = HTRANS == 2'b01;
97      wire trans_nonseq     = HTRANS == 2'b10;
98      wire trans_seq        = HTRANS == 2'b11;
99
100     wire burst_single     = HBURST == 3'b000;
101     wire burst_undef      = HBURST == 3'b001;
102     // No distinction between wrap and incr bursts needed
103     wire burst_quad       = HBURST == 3'b010 || HBURST == 3'b011;
104
105     // For reading/writing data with address from previous clock cycle
106     //wire [7:0] addr_offset_buf;
107     // For triggering state transitions with non-delayed address
108     //wire [7:0] addr_offset;
109     //wire addr_0  = addr_offset_buf == 8'h00;
110     //wire addr_4  = addr_offset_buf == 8'h04;
111     //wire addr_8  = addr_offset_buf == 8'h08;
112     //wire addr_c  = addr_offset_buf == 8'h0C;
113     //wire addr_10 = addr_offset_buf == 8'h10;
114
115     // Integrity check signals to make if statements less tedious.
116     // trans_(non)seq not included as this is different
117     // for entering and during the respective state.
118     wire check_load_iv_init = addr_load_iv_init     && burst_undef  &&  HWRITE;
119     wire check_hashout_init = addr_hashout_init     && burst_undef  && !HWRITE;
120     wire check_status_init  = addr_read_status_init && burst_single && !HWRITE;
121     wire check_sense_init   = addr_sense_init       && burst_single;
122
123     wire check_load_iv = addr_load_iv     && burst_undef  &&  HWRITE;
124     wire check_hashout = addr_hashout     && burst_undef  && !HWRITE;
125     wire check_status  = addr_read_status && burst_single && !HWRITE;
126     wire check_sense   = addr_sense       && burst_single;
127
128     function [2:0] next_state;
129             input dummy; //function requires at least 1 input
130         begin
131             if     (check_load_iv_init) next_state = FSM_LOAD_IV;
132             else if(check_hashout_init) next_state = FSM_HASHOUT;
133             else if(check_status_init)  next_state = FSM_READ_STATUS;
134             else if(check_sense_init)   next_state = FSM_SENSE;
135             else   next_state = FSM_INV_TRANS;
136         end
137     endfunction
138
139
140     //========== COMBINATORIAL BLOCK ==========//
141     always @ (*) begin
142         photon_mat_d   <= photon_mat_q;
143         state_d       <= state_q;
144         photon_init_d <= 1'b0;
145         // delay address reading by 1 cycle to match with data phase
146         addr_buf_d    <= {HADDR[31:24], HADDR[7:0]};
147
148         case (state_q)
149             FSM_SENSE: begin
150                 // Shift out nibbles to LWH interface.
151                 // Reason for including in this state: in SENSE,
152                 // no write action is done to photon_iv.
153                 if(funnel_shift) begin
154                     photon_mat_d <= {photon_mat_q[139:0], funnel_nibble_in};
155                 end
156                 if     (check_load_iv_init && trans_nonseq) begin
157                     state_d <= FSM_LOAD_IV;
158                 end
159                 else if(check_hashout_init && trans_nonseq) begin
160                     state_d <= FSM_HASHOUT;
161                 end
162                 else if(check_status_init  && trans_nonseq) begin
163                     state_d <= FSM_READ_STATUS;
164                 end
165                 else if(check_sense_init   && trans_idle)
166                     state_d <= FSM_SENSE;
167                 else state_d <= FSM_INV_TRANS;
168             end
169
170             // Using this LOAD_IV state instead of indexed-part selection
171             // brings #LUTs down from about 1200 to 400!
172             FSM_LOAD_IV: begin
173                 // 5 beats for 144 bits (4*32 + 1*16), MSB FIRST
174                 if(check_load_iv && trans_seq) begin
175                     state_d <= FSM_LOAD_IV;
176                     photon_mat_d <= {photon_mat_q[111:0], HWDATA};
177                 end
178                 else begin
179                     photon_init_d <= 1'b1;
180                     //advisable to go back to sense here, but not required
181                     state_d <= next_state(1);
182                     photon_mat_d <= {photon_mat_q[127:0], HWDATA[15:0]};
183                 end
184             end
```

```
185
186
187             FSM_HASHOUT: begin
188                 // data operations done with assign, see bottom of file
189                 if(check_hashout && trans_seq) begin
190                     photon_mat_d <= {photon_mat_q[111:0], 32'b0};
191                 end
192                 else begin
193                     state_d <= next_state(1);
194                 end
195             end
196
197             FSM_READ_STATUS: begin
198                 state_d <= next_state(1);
199             end
200
201             //// INVALID ////
202             FSM_INV_TRANS:              //invalid transaction
203                 state_d <= FSM_INV_RESP;
204             FSM_INV_RESP:               //invalid response
205                 state_d <= FSM_SENSE;
206             default:
207                 state_d <= FSM_SENSE;
208         endcase
209     end // end always for state transition
210
211     assign HRDATA = state_q == FSM_READ_STATUS ? {16'hbeef, 15'b0, status} :
212                     state_q == FSM_HASHOUT ?
213                     (
214                         check_hashout ?
215                         photon_mat_q[143:112] : {16'b0, photon_mat_q[143:128]}
216                     ) : 0;
217
218     // ready signal. No bus stall by HREADY = 0 implemented,
219     // because reading data is always completed within one clock cycle.
220     // Reading data chunks can span more clock cycles (bursts),
221     // but no burst needs multiple clock cycles.
222     assign HREADY = 1'b1;
223
224     // responses are always 1'b0 except during two clock cycles
225     // around an invalid transaction
226     assign HRESP = state_q == FSM_INV_TRANS
227                 || state_q == FSM_INV_RESP ? 1'b1 : 1'b0;
228     assign photon_init = photon_init_q;
229     assign funnel_nibble_out = photon_mat_q[143:140];
230 endmodule
```