ICT-56-2020 - Next Generation Internet of Things

D 4.6
Final report on FPGA infrastructure, RISC-V system and accelerators

Document information	
Contract number	957197
Project website	www.vedliot.eu
Dissemination Level	Public
Nature	Report
Contractual Deadline	31.10.2023
Author	Karol Gugala (ANT)
Contributors	Marco Tassemeier (UOS), Mario Porrmann (UOS), René Griessl (UNIBI),
Reviewers	Pedro Trancoso (CHALMERS), Marcelo Pasin (UNINE)

The VEDLIoT project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957197.

Changelog		
V 0.1	2023-07-10	Outline of the document
V 0.2	2023-08-20	SoC generator and AIG description
V 0.3	2023-09-21	FPGA processing infrastructure description
V 0.5	2023-10-05	Version for internal review
V 0.6	2023-10-20	Incorporation of the feedback from the reviewers
V 1.0	2023-10-29	Finalisation

Table of Contents

Executive Summary

In the chapters below, we will provide an overview of the efforts towards enabling the use of efficient low-power accelerators with the SoC system developed within the VEDLIoT project. The document provides a summary of the work done in tasks 4.4, 4.6 and 4.7 within the project. It is the final version of the first report, provided in deliverable D4.3 [1].

In Task 4.4, the reconfigurable communication infrastructure for the RECS systems is developed. Additionally, the basic FPGA infrastructure has been developed, supporting a wide range of different FPGA devices and FPGA boards inside the RECS systems. The architecture especially serves as a basis for the accelerator developments in WP3. With a script-based approach, new accelerators can be easily integrated into the system and the base architecture can be flexibly retargeted to new hardware platforms. Evaluations of the design have been performed using the u.RECS testbed. A special feature of the infrastructure is its support for partial dynamic reconfiguration, which has been evaluated using the Xilinx DPU accelerators from WP3.

In Task 4.6 open source SoC generator software has been extended to support FPGA targets used within the project including hardened ARM CPUs available on AMD/Xilinx chips. Peripheral cores available within the generator have been refactored so that they utilize less FPGA resources and provide better performance in typical AI related tasks (e.g. higher data throughput).

In Task 4.7 an open source system allowing seamless generation of interfaces for instantiating AI accelerators has been developed. The system allows to easily configure and generate data mover interfaces needed to instantiate an accelerator and use it with a bigger system. The Accelerator Interface Generator (AIG) comes with software needed to control the data mover providing the data to and receiving from an accelerator. Example integration of the AIG with the SoC generator improved in T4.6 has been developed and released as open source software on a permissive Apache-2.0 license.

1 Introduction

With the interest in AI technologies rapidly increasing in recent years, there has been a notable increase in demand for AI-related solutions throughout all other technology sectors.

AI-specific hardware is no exception to this trend, with the rise in Edge AI solutions, as they allow for data processing locally, resulting in more efficient use of processing power and safety and privacy risks reduction when compared to AI solutions operating in the cloud.

Developing AI solutions for the edge, however, requires interdisciplinary, often hardware-specific knowledge. This may often render such a pursuit after Edge AI inaccessible, especially to individuals and small companies.

In this light, it should be easy to recognize the importance of making edge AI more accessible by providing open source, vendor independent frameworks that simplify and automate the process of deploying AI models onto edge devices.

Chapter 2 provides an overview of the reconfigurable communication infrastructure of the RECS hardware platforms that are developed in VEDLIoT. Based on this, Chapter 3 focuses on the basic FPGA infrastructure for the cognitive IoT platforms in VEDLIoT. This provides the basis for the integration of FPGA-based machine learning accelerators into the VEDLIoT hardware platforms, especially the newly developed u.RECS. Support for partial dynamic reconfiguration of FPGAs is integrated to increase the flexibility and energy efficiency of the system. This enables the adaptation of accelerators to changing requirements at runtime.

Chapter 4 provides overview of the open source soft SoC generator. The generator can be used for development of the RTL logic for the hardware platforms described in the previous chapter. Following that, chapter 5 describes an open source Accelerator Interface Generator – a project developed within VEDLIoT speeding up integration of memory mapped hardware accelerators with larger systems. Chapter 6 presents examples of SoC generated for both pure FPGA and FPGA SoC systems. The examples show usage of the Accelerator Interface Generator for integrating an accelerator logic.

Chapter 7 concludes the document and is followed by Chapter 8 listing document references.

2 Reconfigurable Communication Infrastructure

In distributed systems, where tasks can be dynamically exchanged between processing systems, the communication needs to be adaptable to the application, providing the requested bandwidth and latency requirements without wasting power for unused capabilities. Therefore, the hardware used in the VEDLIoT project has reconfigurable communication structures. The far edge system u.RECS can be statically reconfigured during setup of the application and the larger RECS servers can dynamically reconfigure their communication structure during run-time. The u.RECS and the t.RECS are the main systems used in VEDLIoT. This chapter shows example setups of communication within these two platforms.

2.1 u.RECS

The u.RECS is a small far edge system for IoT applications. It can host two processing modules and up to two PCIe-based accelerators. The form factor of the processing module can be SMARC 2.1 or NVIDIA Jetson Nano [2]. The FPGA module described in Chapter 2.1.1 can be directly plugged into the SMARC 2.1 slot. In order to populate a second FPGA into the system, an adapter is needed to convert from NVIDIA Jetson Nano to SMARC 2.1 or Xilinx Kria [3]. The result is a multi-FPGA system with multiple ways of communication. The block diagram in Figure 1 shows the two main communication technologies, PCI Express and Gigabit Ethernet.

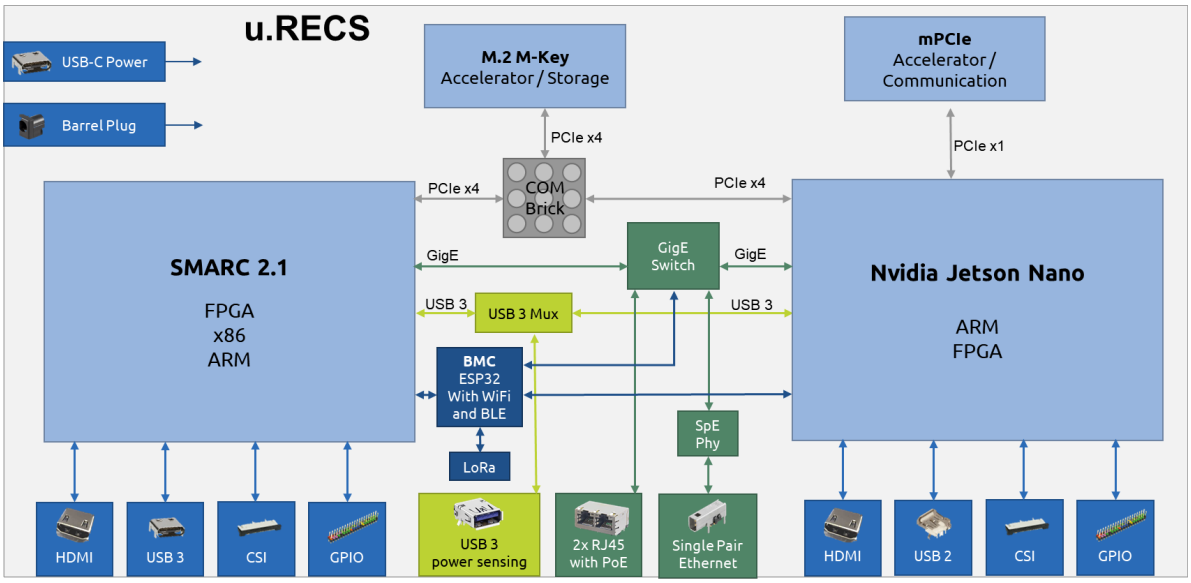

Figure 1 Block diagram of the u.RECS system

Gigabit Ethernet is the basic communication system that connects the compute modules, the BMC and external devices for easy access and data exchange. It is static, but therefore ensures that the modules are always accessible. The PCI Express structure is a flexible infrastructure for determined use cases. In Figure 2, three different COM-Bricks are shown. A COM-Brick is a PCB that can be plugged on the u.RECS, providing a specific routing for the PCI Express lanes.

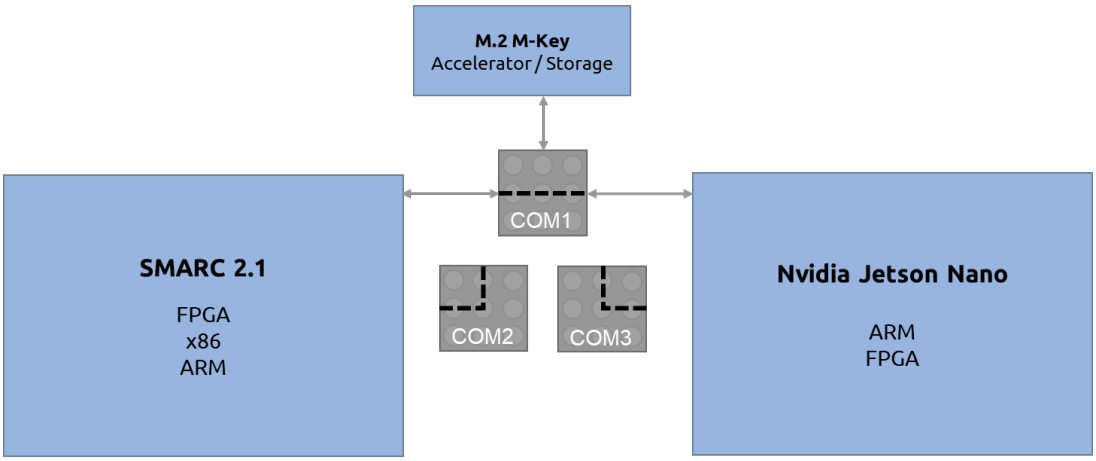

Figure 2 COM-Bricks for high-speed communication

Exchanging these COM-Bricks leads to different PCI Express topologies of the system. For Brick "COM1", the two modules are attached to each other; direct communication is possible if one module can be a PCI Express Endpoint. This is especially true for FPGA-based modules. The Bricks "COM2" and "COM3" are used to attach an M.2-based accelerator to either the SMARC or the NVIDIA module.

A special case is the usage of the "COM1" Brick when two FPGAs are populated. These devices are not bound to the PCI Express communication protocol, they can use lightweight protocols with less overhead. Most of the FPGA vendors support low-level protocols like AURORA [4]. Changing the protocol can improve bandwidth and latency for FPGA-to-FPGA communication.

2.2 t.RECS

The t.RECS is the edge server system of the RECS family. It can host up to three microservers of the new COM-HPC [5] formfactor, two of them are COM-HPC Client and one is COM-HPC Server based. The block diagram in Figure 3 shows the communication infrastructure of the t.RECS. Like for the u.RECS, the main communication is Ethernet based, but in this case it is 10 Gigabit Ethernet.

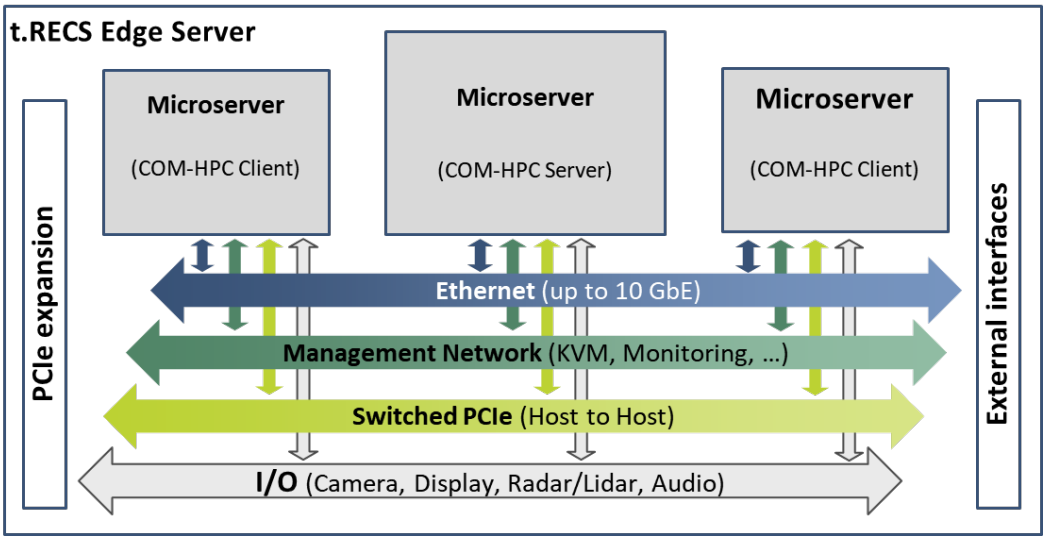

Figure 3 Block diagram of the communication infrastructure of the t.RECS edge server

The high-speed communication of the t.RECS is also based on PCI Express. In Figure 4, possible routing configurations of the PCI Express communication are shown. The standard configuration is shown in part a) of the figure. All microservers are connected using a PCI Express root complex to the PCI Express switch. This is the normal operation mode. It provides virtual network interfaces to the Linux systems and builds up a TCP/IP based communication channel between all connected hosts. In addition, the t.RECS supports direct FPGA-to-FPGA communication as shown in b). FPGAs can be connected in a ring topology using the same low-level protocols as mentioned in the u.RECS section. This direct communication is routed via separate physical lanes. It can be switched on by activating the transceivers in the FPGAs. In this configuration the PCI Express switch is switched off to save power. If the application is in need for even more communication bandwidth, the PCI Express switch can be used in parallel as shown in c). This configuration provides the highest bandwidth but will also consume the most power.

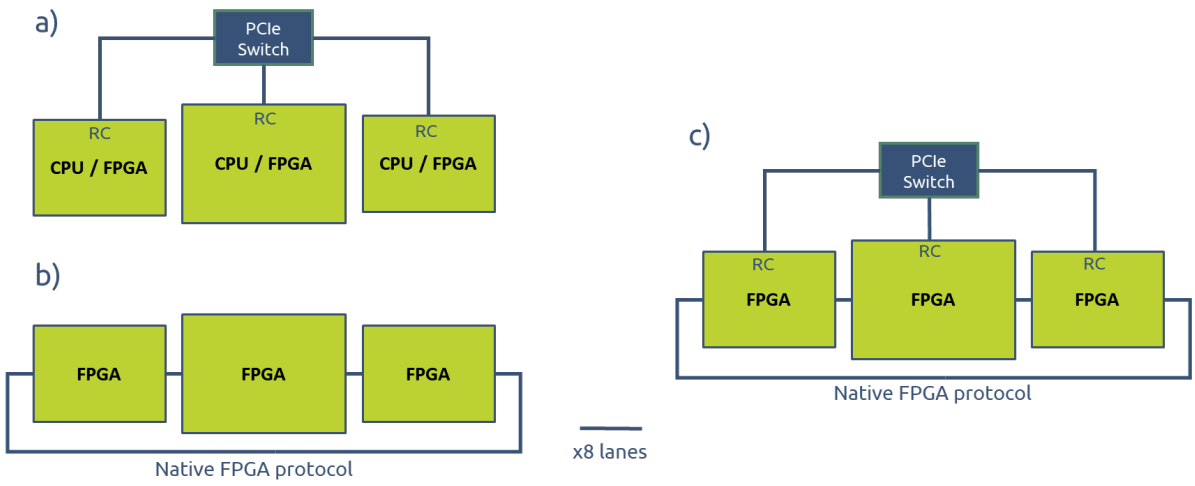

Figure 4 Possible configurations of the PCI Express infrastructure

3 FPGA processing infrastructure for Cognitive IoT

The design of FPGA-based accelerators is one of the key developments in VEDLIoT. This chapter describes the results of Task 4.4, based on the intermediate results discussed in Chapter 2 of deliverable D4.3 [1]. Since D4.3 is a confidential deliverable, parts of the description are repeated here. In Task 4.4, the FPGA processing infrastructure is developed, which serves as a basis for the efficient integration of new accelerators into the u.RECS platform. High-level requirements for the FPGA base infrastructure have been identified from various perspectives:

- From a hardware perspective, support for the RECS platform, mainly the u.RECS system, is the focus, including support for the available I/O interfaces for scalable systems, leading to multi-FPGA architectures. In order to maximize flexibility and reuse of the developments, the design shall be easily retargetable to different FPGAs and systems, first of all the various RECS platforms that are available in VEDLIoT.
- From a software perspective, Linux will be used as the operating system for reconfigurable SoCs with integrated processor cores. This enables easy integration of new interfaces and easy interaction with the hardware accelerators.
- From a development perspective, a block-based design is envisioned, enabling easy adaptation of the architecture as well as easy integration of new ML accelerators without or with minimal changes in the low-level hardware descriptions.
- With respect to resource efficiency, support for dynamic reconfiguration of FPGAs is foreseen, enabling the exchange of hardware components (e.g., ML accelerators) at runtime. This will allow the user to switch between implementations with different power/performance tradeoffs depending on the actual requirements.

This chapter summarizes the architectural concepts, focusing on the developed system architecture for FPGAs, the software architecture and the development environment.

3.1 System Architecture

The FPGA base design developed in VEDLIoT primarily targets the RECS platform described in D4.1 [6], focusing on the u.RECS system. Due to its modular structure, u.RECS supports a wide variety of FPGAs available on SMARC modules [7]. Based on the design decisions, described in D2.2 [8], Xilinx Zynq UltraScale+ MPSoCs will be used as the primary targets for FPGA developments in VEDLIoT.

3.1.1 Hardware Platforms

For the FPGA designs targeting the u.RECS, we focus on SMARC modules, first of all the SECO RUSSELL (formerly codenamed SM-B71). It can be equipped with a wide variety Xilinx Zynq UltraScale+ SoCs, with devices, ranging from ZU2CG to ZU5CG, ZU2EG to ZU5EG, and ZU4EV to ZU5EV MPSoCs [9]. The SMARC module that is used for the base design, described in the following, hosts a Xilinx UltraScale+ ZU4EG. Like all Zynq UltraScale+ devices, the MPSoC integrates a processing system (PS), tightly coupled to an FPGA fabric (PL, programmable logic). The PS comprises a quad-core Arm Cortex-A53 application processing unit, a dual-core Arm Cortex-R5 real-time processing unit and additional units including memory controllers, I/O interfaces and interfaces to the PL. The PL contains 88k 6-input look-up tables (LUTs), 176k Flip-Flops (FFs), 728 DSP Slices, 4.4 Mbit embedded BRAM and 13.5 Mbit embedded URAM. In addition to the MPSoC, the SMARC module provides 2 GByte DDR4-

2400 memory, connected to the processing system and 512 MByte DDR4-2400 memory directly attached to the programmable logic.

While the FPGA design currently focuses on Zynq MPSoCs with integrated Arm processors, the developed design also supports Xilinx FPGAs without embedded processor cores. In this case, a RISC-V processing system based on the soft processing system developed in VEDLIoT can take over the tasks of the Arm PS. This will further increase the flexibility of the system, as discussed in the subsequent chapters.

3.1.2 FPGA SoC Infrastructure

The FPGA architecture has been developed as a block-based design that contains the necessary infrastructure for external communication and hardware accelerator integration. The requirements for external communications interfaces are based on the u.RECS integration. Figure 5 provides an overview of the interfaces that are supported. Some of the interfaces are connected directly to the processing system of the MPSoC while others are implemented in the PL.

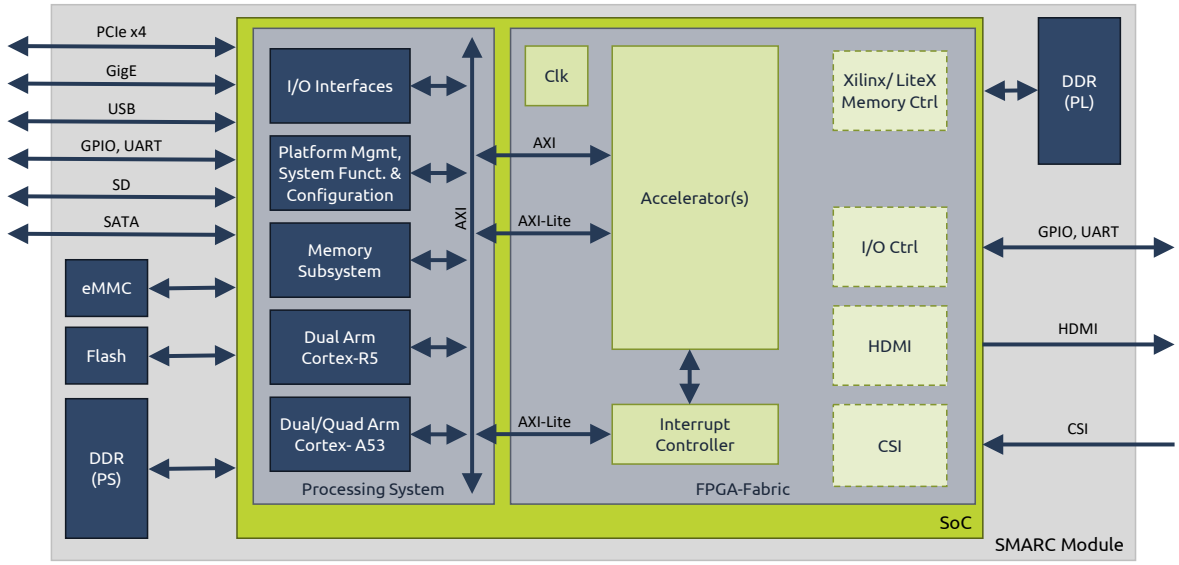

Figure 5 Block diagram of the FPGA base design including the supported interfaces

The PCIe interface connects the different compute modules and accelerators on the u.RECS. With its direct connection to the processing system, this enables a low latency and high-bandwidth connection, e.g., for scalable multi-FPGA systems. Likewise, the Gigabit Ethernet interface is also connected to the PS, allowing for easy integration of network connectivity. USB is also provided with a tight integration into the PS. The interfaces need appropriate drivers for easy access from the applications. Hence, a Linux OS is running on the Arm cores, providing the required software stacks.

In contrast to the interfaces discussed above, the following interfaces are implemented in the FPGA fabric. This enables direct processing of incoming data in the FPGA fabric with minimum latency and without occupying compute time on the processing system. This is particularly useful for handling video data streams, such as received from a camera via the CSI interface or an outgoing video stream, transferred via the HDMI interface. Our design also integrates a memory controller directly connected to the external DDR4 PL memory. Such a controller is especially interesting for accelerators with high memory requirements. This

memory is not shared with the processing system and can be used by accelerators to store intermediate results or parameters for ML accelerators that do not fit in the FPGA internal memory. These interfaces require reconfigurable resources on the FPGA fabric; hence, they are only integrated into the design when required by the application.

For the integration of the ML accelerators, the block design offers an infrastructure consisting of different AXI[1] interfaces, several clock sources with different frequencies and an interrupt controller. The AXI interfaces include AXI-light interfaces which are used to control connected IP-cores and to read and write from and to control registers. Other AXI interfaces are used for high-bandwidth communication with the DDR memory of the processing system via a DMA unit. These can be used for data transfer from the processing system to the FPGA fabric or vice versa. Additionally, both DDR memories (connected to PS and PL) can be used for data exchange between different accelerators.

3.1.3 Dynamic Reconfiguration of Accelerators

An important advantage of FPGAs compared to application-specific integrated circuits (ASICs) is their reconfigurability, enabling highly optimized designs for specific application scenarios (e.g., a specific neural network implementation). However, this reconfigurability comes with a significant overhead in terms of power and gate/interconnect delays when compared to domain-specific architectures, e.g., for machine learning, fabricated in the same technology node. This overhead is significantly reduced by the integration of embedded processors and fixed-function units (like DSP blocks and embedded memories) in modern FPGAs. An additional method to increase the resource efficiency of reconfigurable architectures is partial dynamic reconfiguration. Partial, because only a part of the FPGA design will be exchanged, e.g., an accelerator. Dynamic, because the reconfiguration can be performed at runtime, without interfering with calculations in other parts of the FPGA design. In VEDLIoT, partial dynamic reconfiguration will be utilized in particular to switch between different ML accelerators. Dynamic reconfiguration can be used to enable the system to automatically adapt to changing environmental conditions, like weather changes, when running a neural network on camera data. Another use case for dynamic reconfiguration are mobile systems that run on battery power: based on the current energy budget, accelerators with different power, performance, and accuracy footprints can be selected at runtime. Furthermore, the hardware can be reconfigured to suit an adaption on application level. For instance, we can switch between lane detection and parking assistants in automotive applications.

Dynamic reconfiguration of accelerators can be performed on different granularities. In general, the base design consists of a static part that is only configured once, at power-up, and one or more partially reconfigurable modules. Partial reconfiguration in the VEDLIoT FPGA design can be performed on different levels of granularity, utilizing the toolchain for partial reconfiguration, provided by Xilinx. Three different approaches can be utilized with the developed base design in VEDLIoT:

- A single reconfigurable region
- Multiple reconfigurable regions
- Hierarchically organized reconfigurable regions

[1] Advanced eXtensible Interface

The three approaches provide different levels of flexibility. While a single region can be realized with minimal development overhead, the hierarchical approach provides maximum flexibility and potentially the highest resource efficiency. In the simple case, the design contains one partial reconfiguration region (PR-region) that hosts a single monolithic accelerator design. Partial reconfiguration will exchange the complete accelerator for another one. This makes it necessary to stop the currently running calculations on the accelerator and to disconnect its interfaces from the rest of the design to prevent unintended signals on the connected communication interfaces. The configuration of the hardware in the PR-region can then be overwritten with a new accelerator design. Finally, after the disconnection to the interfaces is released, the calculations on the new accelerator can be initiated, e.g., by the embedded CPU. In this implementation, a PR-region can also be utilized by different individual accelerators, i.e., one big accelerator or several small accelerators can be used. But the important design constraint is, that in this case, all accelerators need to be exchanged; there is no possibility to have individual accelerators up and running, while others are reconfigured.

To circumvent these shortcomings, multiple individual PR-regions can be used to further increase the flexibility of the system. In this case, multiple accelerators can be running in parallel, utilizing individual PR-regions. Therefore, individual accelerators can be replaced at runtime. The process of reconfiguring one accelerator is similar to the process in a design with just one PR-region, as discussed above. The main difference is that since there are individual PR-regions, while reconfiguring one accelerator, the calculations on the other accelerators do not have to be suspended. A drawback compared to the approach utilizing a single PR-region is based on the fact that the number and size of PR-regions is defined at design time. This means that the flexibility of using and exchanging individual small accelerators comes at the cost of constraining the maximum size of the accelerators. This is due to the fact that two or more PR regions cannot be combined to host one accelerator, because internal signals of the accelerator would have to cross the boundaries of PR-regions, which is not supported by the design tools.

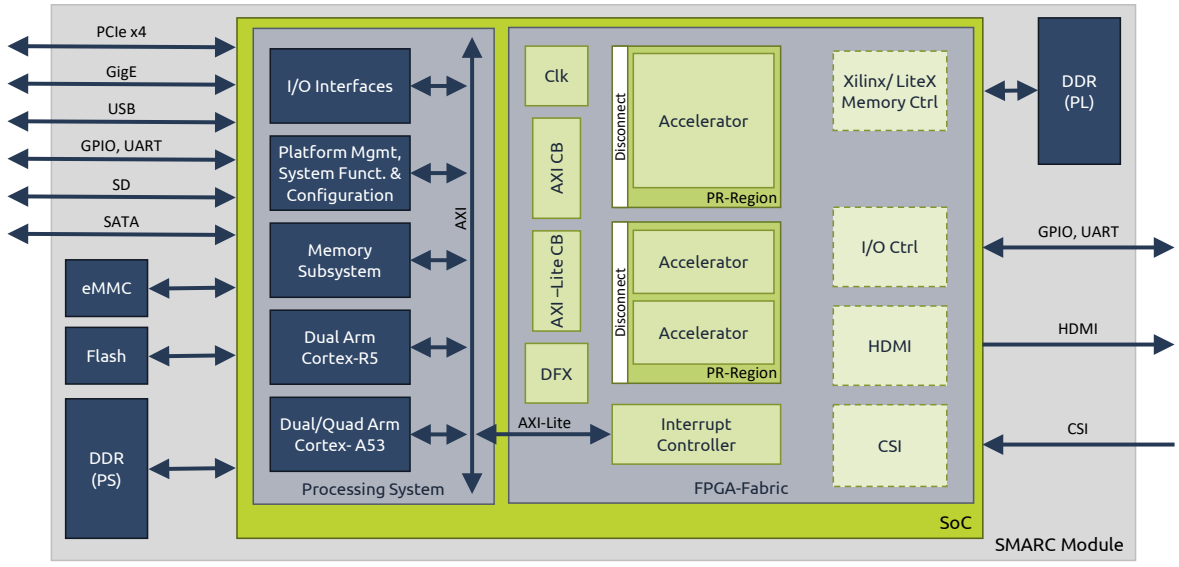

Figure 6 Block diagram of the FPGA base design supporting partial dynamic reconfiguration

Considering the disadvantage of the approach discussed above, hierarchical PR-regions can be integrated in the base design for VEDLIoT, if required. Hierarchical PR-regions allow large

monolithic PR-regions to be separated into smaller Sub-PR-regions with their own interfaces. These Sub-PR-regions can be used individually or can be flexibly combined to host accelerators with larger hardware demands. Figure 6 provides an overview of a base design with two PR-regions, one of which is separated into two Sub-PR-regions.

In the Xilinx tool flow, partial reconfiguration is referred to as Dynamic Function eXchange (DFX). The DFX controller IP, depicted in the block diagram in Figure 6, provides the required management functions. When a trigger for partial reconfiguration occurs, the DFX controller loads the corresponding partial bitstreams from memory and delivers them to the internal configuration access port [10].

3.2 Software Architecture

The Arm cores of the processing system are running PetaLinux, a Linux version for Xilinx SoCs, which is based on Yocto [11]. This Linux environment contains the needed software infrastructure for the I/O interfaces that are directly connected to the processing system, like Gigabit Ethernet, PCIe, and USB. Hence, easy access to the u.RECS platform and its interfaces is provided by means of the common Linux commands. Us-ers can utilize the complete features of the operating system, including, e.g., multi-threading support, which is of vital importance for efficient accelerator integration, as shown in D3.3 [12].

Additionally, the Linux system contains drivers for communication with the FPGA accelerators via AXI. Additionally, the Xilinx runtime libraries offer the possibility to call accelerators via a Xilinx OpenCL API as shown in Figure 7. In this case the data transfers and the calculations are managed by Xilinx OpenCL functions. The code shows an example for the interaction with an accelerator (called Accel_1), requiring start addresses for two arrays, used as data inputs, and one array for the results. After setting the arguments of the accelerator, the processing system schedules three operations: (1) the transfer of the input data to the accelerator memory, (2) the execution of the accelerator, and finally (3) the transfer of the results back to the processing system.

```
// Set in and output buffer for the accelerator
 Accel_1.setArg(0, in1_buf);
 Accel_1.setArg(1, in2_buf);
 Accel_1.setArg(2, out_buf);

 // Schedule transfer of inputs to device memory, execution of the accelerator
 // Transfer of outputs back to host memory
 q.enqueueMigrateMemObjects({in1_buf, in2_buf}, 0);
 q.enqueueTask(Accel_1);
 q.enqueueMigrateMemObjects({out_buf}, CL_MIGRATE_MEM_OBJECT_HOST);

 // Wait for all scheduled operations to finish
 q.finish();
```

Figure 7 Example of calling an accelerator via Xilinx OpenCL functions

As discussed in D3.3 [12], multithreading is often required to efficiently utilize an accelerator, e.g., providing dedicated tasks for pre- and postprocessing. Although this is certainly possible in the provided Linux environment, the development of a multithreaded program poses additional challenges to the developer. To ease the deployment of accelerator implementations, a generic, customizable template for the necessary software parts was developed. It enables the execution of the necessary pre- and post processing steps and calls to

the accelerator using OpenCL calls. The code is generic in the sense that it can be easily adapted to different accelerators. Pre- and post processing functions can be customized by the user as well as the number of threads. Using the template with various parameters for the number of threads and different accelerator implementations enables easy semi-automated design space exploration.

3.3 Development Environment

The base design was created with the Xilinx Vitis Core Development Kit (2022.2) in the Vivado block design environment. It includes all major components of the FPGA infrastructure and external communication interfaces, as discussed above. The block design can be configured to include only the components required for a specific hardware platform or application. The block design is converted to a Vitis platform, easing the integration of custom accelerators and interfaces. The base block design does not contain any accelerator. For creating the Vitis platform, all connections and interfaces that are potentially used by accelerators and other FPGA IP cores, which are integrated into the design in a later step, have to be defined in the base design. This includes the AXI and AXI-light interfaces, the interrupt inputs of the interrupt controller and the different clocks that are provided by the base design. When dynamic reconfiguration of accelerators is foreseen, this base design represents the static design; i.e., this part of the design will only be configured once at power-up and will remain unchanged, regardless of the dynamically loaded accelerators.

As mentioned above, the base design needs to be adapted to match the target application requirements as well as the used FPGA and FPGA platform. This results in a wide variety of different configurations, which can hardly be handled by hand. Therefore, we have developed a scripting environment, managing all aspects of the implementation. All necessary calls to the Vitis build system are automated, enabling an easy transition to new platforms. This is achieved with dedicated configuration files that contain the required information about the targeted hardware platform as well as the application-specific information. The script is based on an Avnet HDL reference design [13]. It automatically builds the complete hardware platform as well as the software infrastructure, including the configuration of the processing system in the base block design and the PetaLinux system. Changes to the FPGA base design, like additional interfaces, located in the FPGA fabric, can be done directly in the script. Configurations of the processing system (e.g., activation and configuration of specific PS interfaces like Ethernet MDIO configuration or DDR memory timings) and I/O constraints of the external interfaces implemented in the FPGA fabric are provided by specific board files for the used FPGA platform. The Linux system is configured by means of a Yocto layer added to PetaLinux build process.

3.4 Example Implementations

Utilizing the approach discussed above, modified designs can be easily built, integrating new accelerators or changing interfaces. Additionally, the designs can be flexibly retargeted to new FPGAs or FPGA boards with different interface connections and external hardware components (e.g., changes in the attached DDR memory). In VEDLIoT, this is especially used for an easy transition between the different FPGA devices in the RECS system.

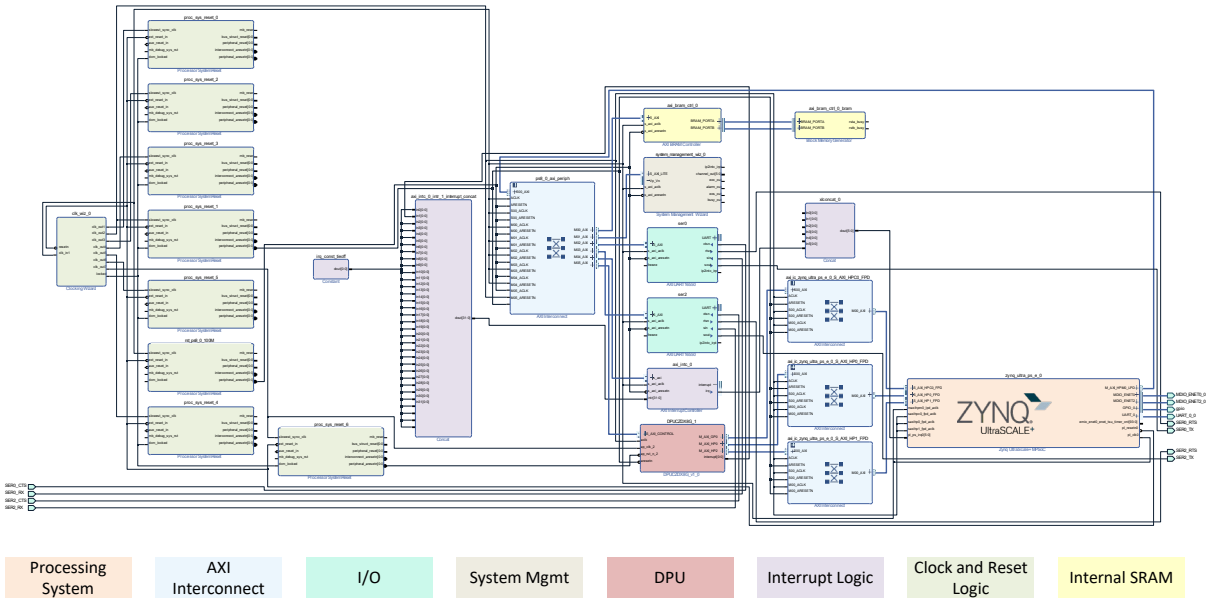

Figure 8 FPGA architecture with integrated DPU

When integrating a new accelerator, the connections of the different interfaces of the accelerator to the corresponding interfaces of the base design are done automatically. Some interfaces can be connected to different targets, like the clock, for which different clock frequencies can be chosen. In this case, a configuration file is used to define the intended connections. Figure 8 shows a base design, integrating an accelerator based on the Xilinx Deep Learning Processor Unit (DPU), which is discussed in detail in D3.2 [14]. For illustration, the components of the base design have been grouped in colours.

The DPU is connected to the processing system using three AXI connections for data and instruction exchange and an additional AXI-Lite interface for controlling the DPU. If needed, for example, by other accelerators, additional AXI interconnects are automatically added to the design. Additional signals that are connected to the DPU are two clocks, which are required by the IP core and the corresponding reset signals. For this example, the number of additional system components is minimized in order to maximize the available resources for DPU integration. For external communication, two UART IP cores are integrated in addition to the Gigabit Ethernet interface of the processing system. The latter is primarily used for communication with the board during performance evaluation. The interrupt logic multiplexes the various interrupt sources of the design. The integrated system management IP core also enables on-chip voltage and temperature measurements. Embedded SRAM with configurable size can be used for internal data storage.

The accelerators can be flexibly exchanged supporting the accelerators developed in VEDLIoT as well as other open-source accelerators or different variants of the Xilinx DPU. As an example, Table 1 provides an overview of the resource requirements of the base design, integrating three different DPU configurations, namely B512, B2304, and B4096. All designs have been implemented for a DPU clock frequency of 300 MHz. Additional details on the performance of the implementations are provided in deliverable D3.2 [14]. As can be seen, the base design requires less than 10% of the available look-up tables (LUTs) and flip-flops (FFs). For the embedded SRAM, four BRAM blocks are used; the size of this memory can be configured by the user. No embedded UltraRAM (URAM) and DSP blocks are used by the base design.

Table 1 Resource requirements of the base design in combination with three DPU configurations, implemented on Xilinx Zynq UltraScale+ ZU4EG

		DPU Configuration					
Resources		**B512**		**B2304**		**B4096**	
Complete Design	LUTs	34,456	39.2%	47,107	53.6%	56,685	64.5%
	FFs	43,557	24.8%	78,215	44.5%	107,732	61.3%
	DSPs	110	15.1%	422	58.0%	690	94.8%
	BRAMs	13.5	10.5%	61	47.7%	81	63.3%
	URAMs	16	33.3%	40	83.3%	48	100%
Base Design	LUTs	8,439	9.6%	8,434	9.6%	8,456	9.6%
	FFs	10,205	5.8%	10,205	5.8%	10,205	5.8%
	DSPs	0	0%	0	0%	0	0%
	BRAMs	4	3.1%	4	3.1%	4	3.1%
	URAMs	0	0%	0	0%	0	0%
DPU	LUTs	26,017	29.6%	38,673	44.0%	48,229	54.9%
	FFs	33,352	19.0%	68,010	38.7%	97,527	55.5%
	DSPs	110	15.1%	422	58.0%	690	94.8%
	BRAMs	9.5	7.4%	57	44.5%	77	60.2%
	URAMs	16	33.3%	40	83.3%	48	100%

An example of a dynamically reconfigurable design, implementing the base design with support for runtime reconfiguration and a B512 DPU placed in the partially reconfigurable region, is depicted in Figure 9. In the layout, the PR region is highlighted in blue; PS and static design are marked in green. At runtime, the DPU can be exchanged with another accelerator, providing a different trade-off for power, performance and accuracy by reconfiguring the PR region.

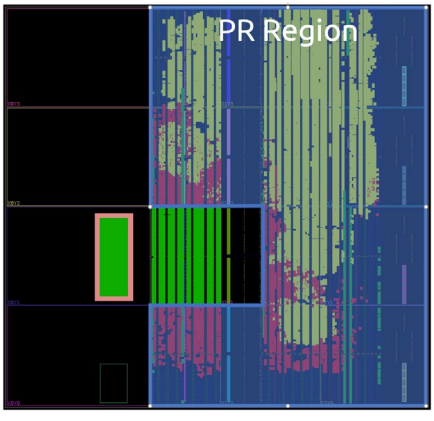

Figure 9 Example of the FPGA-implementation of the base design including a B512 DPU on an UltraScale+ ZU4EG

In addition to the Ultrascale+ MPSoCs discussed so far, the base design also supports the new Xilinx Versal devices. Besides an Arm processing system and an FPGA fabric, this new architecture also includes a matrix of AI engines. These AI engines are vector processing units organized in an array connecting them to each other and to other parts of the SoC. For on-chip communication, the Versal architecture includes a network on chip (NoC) which connects the processing system, the programable logic, the AI engines, and IO components. It enables high bandwidth connections between all major components reducing data transfer bottlenecks. This new architecture promises good performance, especially for ML applications utilizing AI engines. Test results running the VEDLIoT benchmarks on the dedicated Versal DPU, as presented in D3.3 [12], show performance, power and accuracy results comparable to embedded GPUs. Especially the YoloV4 results make the Versal architecture a promising candidate for the smart home use case in VEDLIoT. Figure 10 shows an example of the base architecture on a Xilinx Versal. This implementation enables two accelerators running in parallel, both utilizing FPGA resources and AI engines.

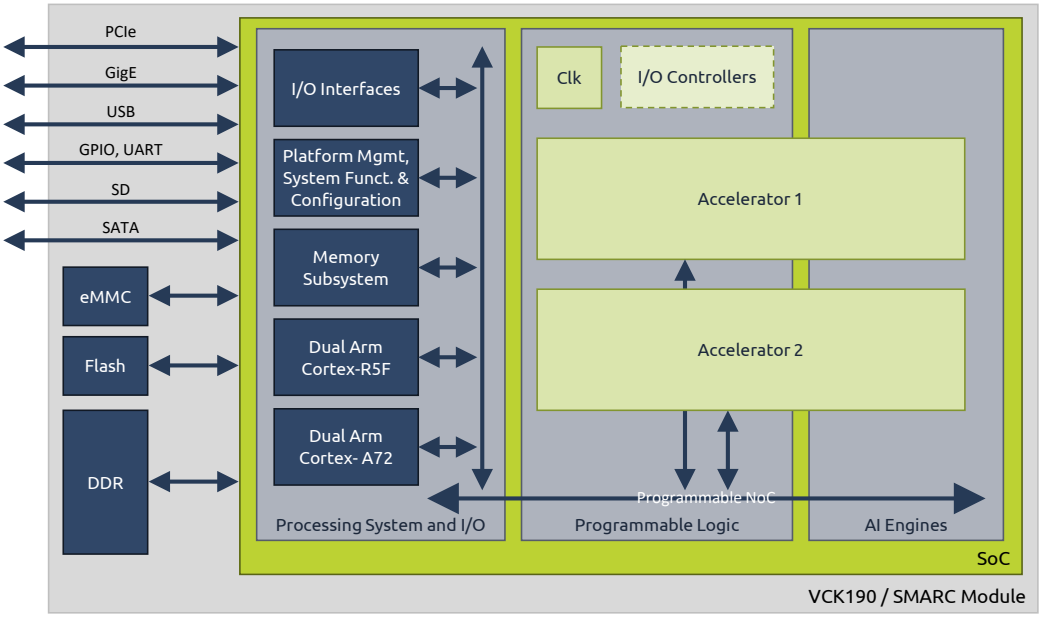

Figure 10 Block diagram of a base design on Xilinx Versal proving integrating two accelerators

4 SoC generator overview

The work in the T4.6 and T4.7 of the VEDLIoT project focused on providing software allowing design and configuration of an SoC that can be used to process the ML payloads on the FPGAs. In designing the FPGA soft SoC system, we used LiteX, an open source system-on-chip (SoC) generator and system builder that enables the creation of custom SoCs tailored to specific requirements. It provides a flexible and modular framework for designing and generating complex digital systems, allowing users to create their own SoC designs from scratch or modify existing ones, also for mixed-language projects.

At the core of the generator is the concept of Intellectual Property (IP) cores, which are pre-designed and verified hardware modules that can be integrated into the SoC. These cores include processors, peripherals, memory controllers, communication interfaces, and other. The generator offers a variety of cores to choose from, allowing developers to select the ones that best suit their particular design.

The SoC builder supports multiple CPU architectures, including RISC-V, OpenRISC, LM32, Zynq, etc., providing options for both open source and proprietary instruction sets. Additionally, it offers support for different system interconnects, allowing efficient communication between various components within the SoC.

Users can configure the desired components and connectivity options of their SoC, e.g. select the required IP cores, set the desired parameters, define memory maps, or establish interconnects between different components.

4.1 External components integration

In an SoC, components are connected with buses which serve as a communication system. Through a bus, components are able to transfer data (data bus) and exchange information about requests (control bus). Such information will most often consist of read/write requests and their return status.

As a means of arranging access to all components connected to a single bus, each of the components is assigned an address. When connecting two similar (or dissimilar) buses, a bridge is placed. Bridges, like other components, are assigned their individual addresses as well.

As an example, let us take a component that is connected to a bus at the `0x1` address. This bus is connected through a bridge at the `0x1000` address to the main bus. In order to communicate with this component from the main bus, we need to send a request (through a selected communication protocol) to the `0x1001` address. The main bus will then delegate this request to the bridge, which will later decode the request's address as 0x1 and forward it to the component.

In the SoC generator, the *SoCBusHandler* class, accessed through *SoCCore.bus*, is responsible for communication between the CPU and the rest of the system.

Therefore, in order to manually incorporate SoC components in LiteX, one needs to connect a component's control bus to this bus. This, usually, can be done through the *SoCCore.add_slave* method.

To use a Verilog component, first, you need to add a source file to the chosen platform.

```
class Component(Module):
      def __init__(self, platform):
            platform.add_sources(path/to/Component.v)
```

Then, proceed with instantiating interfaces (or signals) available from *Component.v*.

```
            self.axi = AXIInterface(data_width=data_width)
            self.axi_csr = AXILiteInterface(data_width=data_width)
            self.irq_ex = Signal(1)
```

You can connect the interfaces listed above to the component via *Instance* by simply assigning corresponding signals.

```
             self.specials += Instance("ComponentTop",
                               i_clock = ClockSignal("sys"),
                               i_reset = ResetSignal("sys"),
                               o_io_axi_aw_id = self.axi.aw.id,
                               o_io_axi_aw_awaddr = self.axi.aw.addr,
                               o_io_axi_aw_awlen = self.axi.aw.len,
                                o_io_axi_aw_awsize = self.axi.aw.size,
                                o_io_axi_aw_awburst = self.axi.aw.burst,
                                                    ...
                              i_io_csr_ar_araddr = self.axi_csr.ar.addr,
                              i_io_csr_ar_arprot = self.axi_csr.ar.prot,
                              i_io_csr_ar_arvalid = self.axi_csr.ar.valid,
                               o_io_csr_ar_arready = self.axi_csr.ar.ready,
                                           ...
                              o_io_irq_ex = self.irq_ex
)
```

You can then connect a component to the main bus by declaring the component's address space via *SoCRegion* and connecting its Control and Status Register (CSR) handling bus to the target's main bus. The main bus standard is imposed by the chosen architecture.

```
class BaseSoC(SoCCore):
      def __init__(self, **kwargs):
            platform = chosen_platform.Platform()
            SoCCore.__init__(self, platform, **kwargs)

            ...

            self.submodules.Component = Component(platform)
            core_region = SoCRegion(origin=0xbase_addr, size=0xsize)
            self.add_slave("Component", self.axi_csr, core_region)
```

If, apart from CSRs, the component also transfers data, the appropriate bus interface needs to be instantiated and connected to the interface provided by the chosen target (in case of Zynq7000 it's *axi_hp_slave*).

```
self.comb += self.AIG.axi.connect(self.cpu.add_axi_hp_slave())
```

If supporting interrupts, they can be instantiated with an architecture-specific method.

```
self.cpu.cpu_params.update({"i_IRQ_F2P" : self.irq_ex})
```

It is worth noting that connecting buses of different type (i.e. Wishbone with AXI protocol) will result in additional converting logic.

4.2 Accelerator integration

Within the VEDLIoT T4.6 and T4.7 tasks, we are exploring two approaches of interfacing with accelerators:

- Accelerator connected to a system bus consuming data from the memory and writing results back to it.
- Tight integration with a CPU where the accelerator is implemented as a Custom Function Unit and is driven with custom CPU instructions.

4.2.1 Accelerator on the bus

Having introduced SoC development tools, the remaining component is the integration of the given accelerator to the rest of the SoC.

One solution is to connect an accelerator, alongside two data movers, to the system bus. Those data movers would facilitate the read and write operations between the accelerator and the main memory.

On top of the data transfers, it is crucial to incorporate a Control/Status Register (CSR) handling component for this system to work effortlessly - as those later allow the accelerator-system to communicate with dedicated software, such as device drivers.

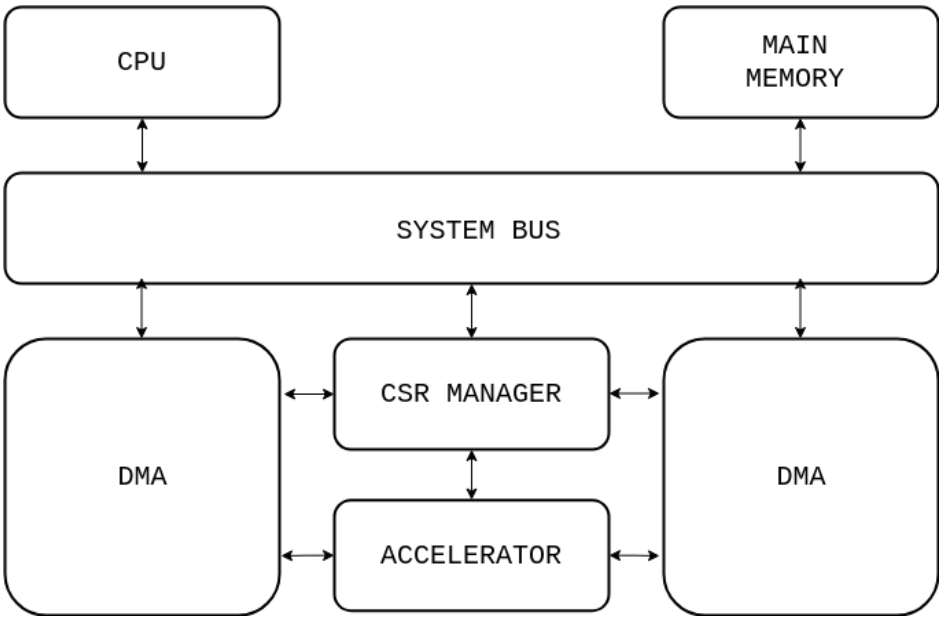

Figure 11 Accelerator on the bus diagram

On top of the data transfers, it is crucial to incorporate a Control/Status Register (CSR) handling component for this system to work effortlessly - as they later allow the accelerator-system to communicate with dedicated software, such as device drivers. Apache VTA or Nvidia's NVDLA can serve as an example of an open source accelerator that is implemented using this method.

4.2.2 RISC-V tightly coupled accelerator

Another approach of interfacing with accelerators we are exploring as part of VEDLIoT Is tight integration with a RISC-V CPU, where the accelerator is implemented as a Custom Function Unit (CFU) and driven using custom CPU instructions. It adds a custom instruction to the ISA using a standardized format defined by the CFU working group of RISC-V International.

5 Configurable Accelerator Interface Generator

In this section, we concentrate on describing the mechanisms behind the Accelerator Interface Generator (AIG), a configurable, vendor-independent tool that simplifies the process of, amongst others, development of IoT devices. AIG incorporates a custom AI accelerator design into the system by connecting it, alongside two data movers, to the system bus. In this particular case, the role of data movers can be performed by a Direct Memory Access (DMA) controller, which allows for RAM access independently from the CPU.

In order to manage the accelerator and DMA CSRs simultaneously, both the DMAs and an accelerator can be connected to the CSR decoder which later is connected to the system bus.

To further improve the data exchange between DMAs and the AI accelerator, the components can be internally connected with an AXI-Stream interface - a simple and efficient data-transferring protocol.

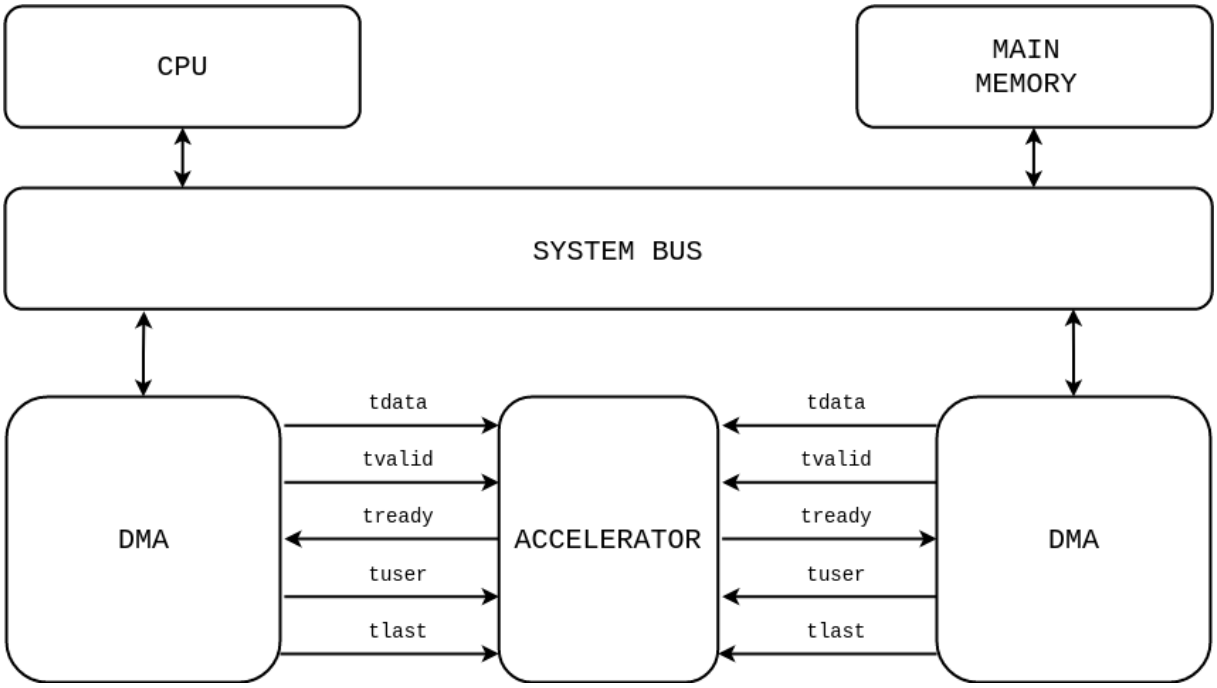

Figure 12 DMAs and accelerator internally connected via an AXI-Stream interface

We can find a similar AI accelerator architecture approach in the VTA extension in the Apache TVM framework as well as the NVIDIA Deep Learning Accelerator (NVDLA) from NVIDIA. However, for our needs, we discovered that applying these solutions would require numerous alterations for the SoC to be applicable on different platforms and in different configurations, as both of the architectures assume the use of their dedicated software.

5.1 FastVDMA

For the purpose of using an AI accelerator on the system bus, we used Antmicro's open source DMA controller – Fast Versatile DMA (FastVDMA). FastVDMA is vastly customizable, allowing modification of various bus parameters such as address, data width or maximum

burst. It can easily be generated in different configurations of supported data and control buses, providing compatibility with the following interfaces:

Data:

- AXI Stream
- AXI4
- Wishbone

Control:

- AXI4 Lite
- Wishbone

Recent improvements to the FastVDMA core introduced a new level of configurability, where the required interface can be generated on-the-fly. Also a single instance of the core uses unique block names to prevent naming collisions in bigger systems and enable generating more complex systems with multiple FastVDMA cores of various configurations.

5.2 Demosaicer cores as an example

In order to demonstrate the functionality of the Accelerator Interface Generator a simple stream processing core has been integrated with the system. While the core itself is not an AI computations accelerator, it is simple enough to show the integration process. Also, video processing cores are often used in edge vision systems to accelerate preprocessing of the raw video frames garnered from the sensors.

The below is an example use case FPGA-based demosaicing system that converts raw data obtained from CCD or CMOS sensors and reconstructs the image using three different interpolation algorithms controlled via a dedicated wrapper. The three interpolation methods are:

- Nearest neighbor interpolation, where the nearest pixel is used to approximate the color value. This algorithm is simple to implement and is common in real-time 3D rendering. It uses a 2x2 px matrix and is the lightest and easiest method to implement.
- Bilinear interpolation, which establishes color intensity by calculating the average value of the 4 nearest pixels located diagonally in relation to the given pixel. This method uses a 3x3 px matrix and gives better results than the nearest neighbor interpolation method but takes up more FPGA resources.
- Edge directed interpolation, which calculates the pixel components in a similar way to bilinear interpolation, but uses edge detection with a 5x5 px matrix. This algorithm is the most sophisticated of the three, but gives the best results and eliminates zippering.

5.3 Accelerator Interface Generator

The Accelerator Interface Generator (AIG) aims for configurability and simplicity. Its main goal is to create a generic accelerator interface that could be used in efficient Deep Learning in IoT.

The AIG is a vendor-independent project that allows for generation of a Verilog module that incorporates two DMAs and an user-defined accelerator. The only requirement for the accelerator is compatibility with AXI Stream.

AIG supports custom CSRs, which can be defined via a configuration file.

AIG provides functionality to generate example SoC generator target boards, including accelerator design.

5.3.1 Generating bitstream from target device

Produced target device description - *aig_generated_target.py* is a structured SoC description that can also be processed by the SoC generator.

Having the SoC generator and its dependencies in path, it is sufficient to execute the generated script with *--build* option.

```
python aig_generated_target.py --build
```

By default, the toolchain used for synthesis, implementation, place & route is Vivado.

Upon successful execution, all of the Vivado produced files with the design source code will be available within the build directory.

```
build/aig_generated_target/gateware/
```

This directory will also contain the generated bitstream which then can be loaded onto the device via i.e. openFPGALoader or written to the pre-formatted SD card.

5.3.2 Customization

The AIG configuration is stored in a JSON file format and it's validated using a JSON Schema specification.

AIG allows users to define custom CSRs with the configuration file. The implementation is generic by nature and can be fully customized.

All possible customizations can be specified in the configuration file, including:

- Address Space - via *baseAddress* fields for each component. The AIG uses a custom decoder that grants access upon those values.
- FastVDMA parameters (optional)
  - Address and Data Widths
  - Max burst size
  - FIFO depth
- Accelerator description
  - Source file
  - Signals: I/O AXI Stream, clock and reset
  - Custom CSRs (optional)

AIG implements three different types of CSRs:

- Status Register - read-only,

- Auto-Clearing Register - read-write; overriding the data once write request has finished,
- Storage Register - read-write; overrides with each write. Each CSR can be specified with:
    - name (unique),
    - type,
    - address (which a absolute address in the SoC address space),
    - fields (which names need to be the same as in the accelerator's top Verilog module to be properly connected).

What is important, the code of both the DMA and the CSRs is vendor agnostic and can therefore be easily used with any FPGA or included in ASIC design.

5.4 Test system overview and tests

The plot below depicts the throughput of the AIG system on hardware being clocked at 100 MHz on both Arty A7 and Zynq Video Board. The experiment has been conducted with a full system setup - target generated in the SoC generator, Linux v5.15 with FastVDMA and FPGA ISP provided software and the v4l2-ctl tool.

The AIG system has been tested in four different configurations. That included using AXI4 or Wishbone for DMA input/output buses and Wishbone or AXI4-Lite for CSR handling.

Each record in the plot represents the average throughput from 3000 frames sent (total of around ~3000 MB in blocks of size 238.8kB and 955.2kB), 1000 for each FPGA ISP algorithm - the differences between efficiency of algorithms were negligible.

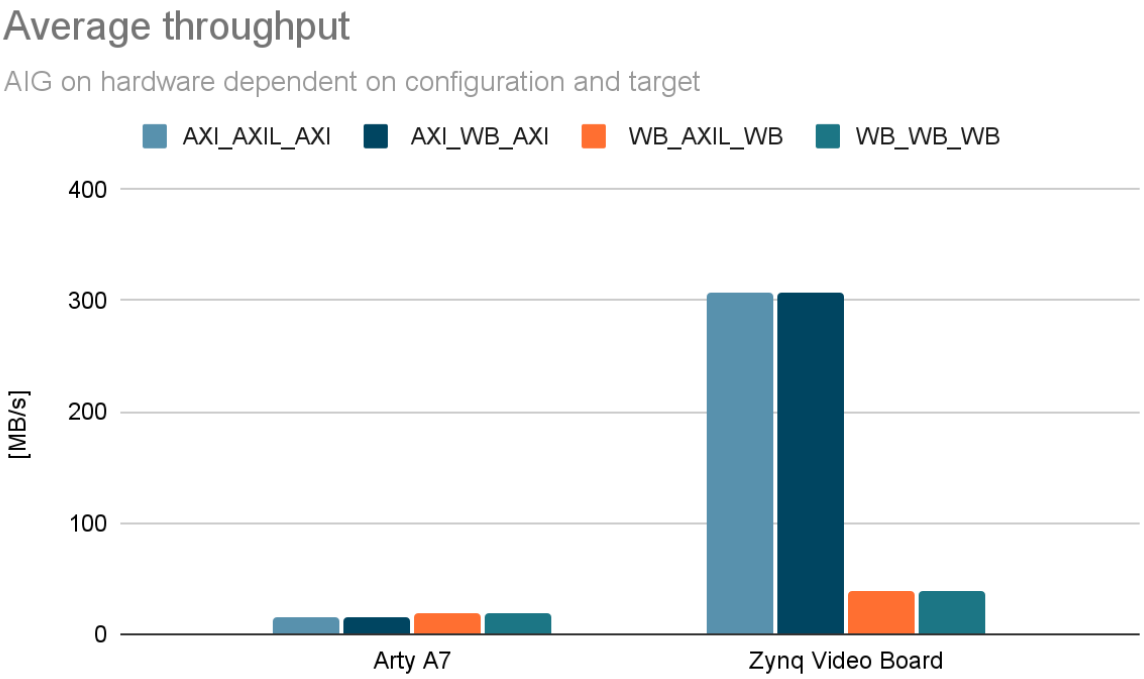

Figure 13 Accelerator interface data throughput

The AIG uses chiseltest for testing its components individually.

The AIG provides the generic AIGInterface test class for Cocotb that chooses configuration-appropriate Cocotb interfaces alongside suitable memory definitions and read/write file methods.

An example test suite is provided for the FPGA ISP demosaicing cores, where a picture in Bayer format is passed to the accelerator and a RGB decoded picture is passed to the output and saved to the file.

6 VEDLIoT SoC

VEDLIoT SoC cannot be considered as a single implementation that is given to users, but rather a set of building blocks along with a generator and build system allowing defining an SoC structure, generating required code and build the SoC for a selected target

The SoC generator is using migen domain specific language and LiteX library to define the target platform. It integrates the Accelerator Interface Generator, described in section 5 enabling users to integrate their own accelerators in the system.

A few examples of the system have been provided to demonstrate the possibilities and give future users an entry point to start working with the design and be able to instantiate their accelerators.

The SoC generator developed within T4.6 and T4.7 of the VEDLIoT project can be used with the hardware developed in other tasks of the project. The highly configurable nature of the FPGA based processing infrastructure requires highly configurable software and RTL code allowing adaptations of the processing flow depending on the payload. The SoC generator and Accelerator Interface Generator described in the previous sections were developed to address exactly this requirement. Both parts were released on permissive open source licenses, so they can also be used outside the VEDLIoT project.

6.1 Example targets

The SoC projects provides example build targets for the following platforms:

- Zynq Video Board with Zynq 7000 SoC
- Arty A7 (with open source RISC-V CPU)

The platforms were chosen to present the system functionalities with a hardened ARM cpu and soft RISC-V CPU. The generator/build floe, however, provides the infrastructure that facilitates adding support for custom target devices.

6.1.1 Zynq Video Board

Zynq Video Board is an open hardware carrier board designed for Enclustra Mars SoC Modules. Containing two MIPI CSI-2 camera interfaces, D-PHY receiver and Pmod connector it can serve well for applications on edge oriented around image processing.

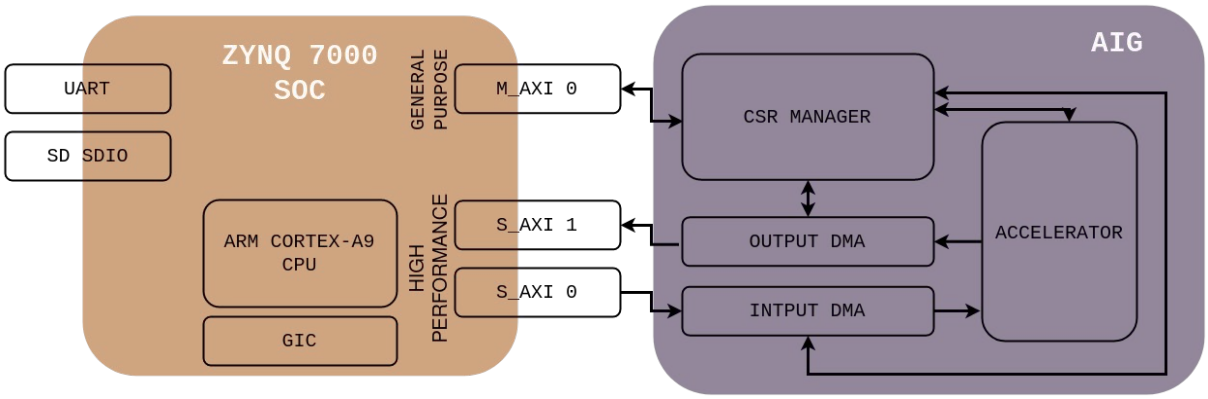

Figure 14 AIG system connected to Zynq 7000 SoC on Zynq Video Board

When deploying an application onto the Zynq Video Board, AIG will utilize high performance AXI buses provided by Zynq 7000 SoC for more efficient data transfers between the DMAs and RAM.

Control communication will be carried out through general purpose AXI interface.

Aside from the ports visible in Figure 14, Zynq's Processing System 7 offers a range of MIO ports that can be utilized for connecting USB, CAN, I2C, SPI or ethernet peripherals as well as the extended ones that can be repurposed, additionally supporting i.e. JTAG.

In order to generate Zynq Video Board with AIG and example accelerator target device description, you may proceed with provided example with:

```
exportTARGET=antmicro_zynq_video_board
export CONFIG_FILE=examples/configs/config_zvb.json
make target
```

This will produce *aig_generated_target.py,* which is the target device description, as well as *AIGTop.v* and *fpga_isp.v* that are Verilog source code files that will be later needed for synthesis.

6.1.2 Arty A7

Arty A7 is a development platform designed around Artix7 FPGA. It was developed specifically to facilitate a soft processing system, which could be altered and fitted to individual applications.

For the needs of the AIG project, the VexRiscV SMP CPU was utilized - an open source RISC V implementation.

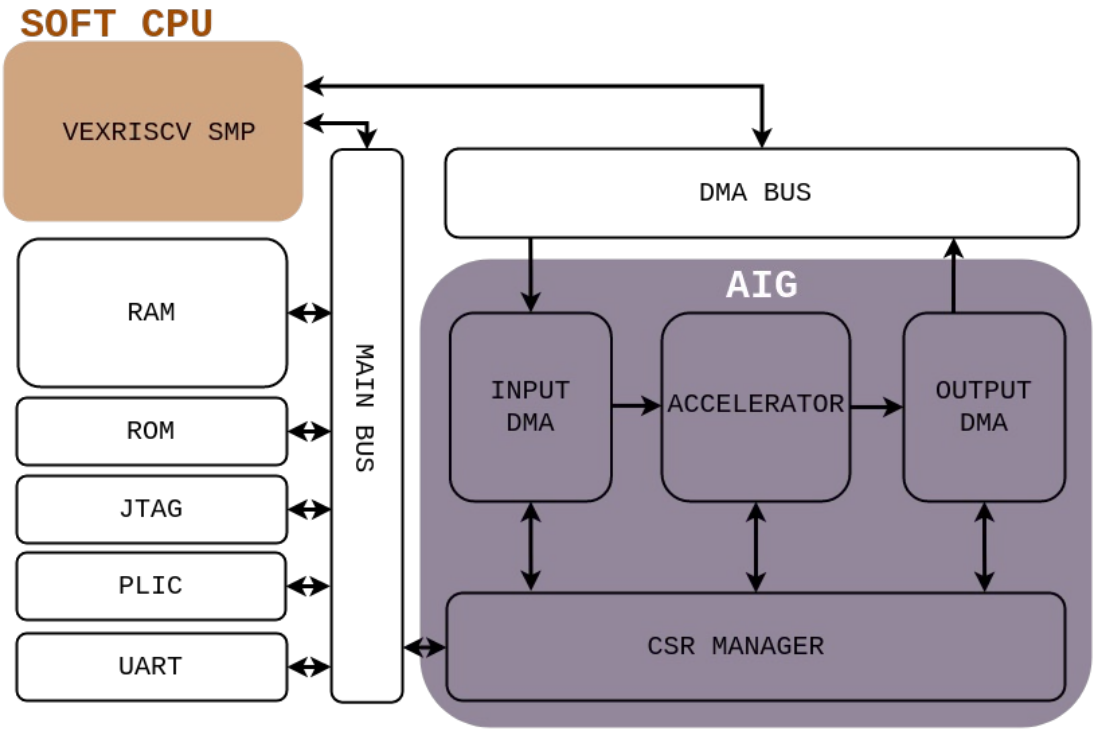

Figure 15 AIG system ported onto Arty A7 with VexRiscV SMP - soft CPU

In the case of the Zynq Video Board, a great number of components were already embedded within the Zynq SoC. Here, since the soft CPU is utilized, some of them need to be added manually, such as RAM, UART or platform interrupt controller.

All of those components are connected to the main bus. VexRiscV provides wishbone interfaces for both control and data transfers. The DMAs will access RAM through *dma_bus*.

Similarly with Zynq Video Board target device, one may use the provided example to generate Arty A7 with AIG description.

```
Export TARGET=digilent_arty
export CONFIG_FILE=examples/configs/config_arty.json
make target
```

This will produce *aig_generated_target.py,* which is the target device description, alongside *AIGTop.v* and *fpga_isp.v* sources.

7 Conclusion

This document provides an overview of the accomplishments achieved in the VEDLIoT project concerning FPGA infrastructure and software for the development, construction, and deployment of logic designs onto hardware. A significant portion of the project concentrated on enhancing the hardware platform to seamlessly integrate with larger systems. This involved the expansion of the u.RECS IoT platform to include FPGA-based devices and the creation of FPGA-based nodes for the t.RECS system, facilitating the offloading of server-side processing to dedicated logic running within the FPGA.

An FPGA base infrastructure supporting FPGAs from u.RECS and t.RECS was implemented to enhance flexibility and ease of use. The design is crafted to be easily retargetable across different FPGAs and RECS platforms featured in VEDLIoT. Furthermore, resource efficiency is a central concern, with plans for dynamic FPGA and communication reconfiguration to facilitate the swapping of hardware components, such as machine learning accelerators, at runtime.

The RTL generator software was a focal point, tailored to produce System-on-Chip (SoC) designs optimized for specific FPGA families. Users can select from a range of available RISC-V soft cores and robust ARM cores found in FPGA platforms like Zynq-7000 and Zynq Ultrascale plus MPSoC to be included in the system.

To enable the smooth integration of memory-mapped accelerators with the generated SoC, an additional tool known as the Accelerator Interface Generator was developed. This open-source software simplifies the process of generating the necessary digital logic and software for integrating a memory-mapped accelerator.

The efforts undertaken in the VEDLIoT project, as outlined in this report, have led to an expansion of the processing capabilities of the t.RECS system and the creation of innovative u.RECS devices. Furthermore, the open-source software and RTL developed in this context open up new possibilities for the hardware platform and can have broader applications beyond the scope of the VEDLIoT project.

8 References

[1] VEDLIoT, „Deliverable D4.3 First report on FPGA infrastructure, RISC-V system and accelerators," 2022.

[2] Nvidia, "Jetson Xavier NX and Jetson Nano Interface Comparison and Migration Application Note, V1.2, 2021/12/10".

[3] Xilinx, „DS987 (v1.2) Kria K26 SOM Data Sheet," March 15, 2022.

[4] Xilinx, "Aurora 64B/66B v12.0 LogiCORE IP Product Guide," [Online]. Available: https://docs.xilinx.com/r/en-US/pg074-aurora-64b66b.

[5] PICMG, „COM-HPC Overview," [Online]. Available: https://www.picmg.org/openstandards/com-hpc/.

[6] VEDLIoT, "Deliverable D4.1 First report on cognitive IoT hardware platform and microserver development," 2022.

[7] SGeT - Standardization Group for Embedded Technologies, "Smart Mobility ARChitecture (SMARC)," [Online]. Available: https://sget.org/standards/smarc/.

[8] VEDLIoT, "Deliverable D2.2 Hardware platform architecture," 2021.

[9] Seco, "SM-B71 - SMARC Rel. 2.0 compliant module with the Xilinx Zynq Ultrascale+™ MPSoC".

[10] Xilinx , "Vivado Design Suite User Guide - Dynamic Function eXchange - UG909," February 2022.

[11] Xilinx, "UG1144 - PetaLinux Tools Documentation: Reference Guide (UG1144) (v2022.1)," 2022.

[12] VEDLIoT, „Deliverable D3.3 Evaluation of the DL accelerator designs," 2022.

[13] Avnet, "Avnet HDL Reference Designs," [Online]. Available: https://github.com/Avnet/vitis.

[14] VEDLIoT, "Deliverable D3.2 Initial report on the DL accelerator design," 2022.

9 List of Figures

10 List of Tables