Project:	ISOLDE: customizable Instruction Sets and Open Leveraged Designs of Embedded riscv processors
Reference number:	101112274
Project duration:	01.05.2023 - 30.04.2026
Work Package:	WP5: Use Cases and Demonstrators
Deliverable	**D5.1**
Title	**Description of demonstrators architecture**
Type of deliverable:	Report
Deadline:	30.04.2024
Creation date:	22.11.2023
Authors:	Antonio Sciarappa, LDO Holger Blasum, SYSGO Samuel Ardaya-Lieb, CONS Danut Rotar, CAR Esther Soriano, FENTISS Davide Di Ienno, TASI Cyril Koenig, ETHZ Wolfgang Ecker, IFAG Dominik Riemer, BYK Philipp Zehnder, BYK Catalin Ciobanu, IMT Daniele Jahier Pagliari, POLITO Alessio Burrello, POLITO Josep Jorba, Rapita Systems, SL Michael Gautschi, ACP Maurizio Martina, POLITO George Suciu, BEIA Anais Sachian, BEIA Frank Oppenheimer, OFFIS
Involved partners:	LDO, CAR, SYSGO, ACP, CONS, FENTISS, TASI, ETHZ, IFAG, BYK, IMT, RAPITA, SAL, ACP, POLITO, OFFIS, POLIMI, UNIBO, BEIA
Contacts:	Antonio Sciarappa, LDO, antonio.sciarappa@leonardo.com
Reviewers:	E4, INTEL, IFX, CAR; ETHZ, LDO

Change Records

ISSUE	DATE	§ CHANGE RECORDS	AUTHOR
Issue 1	29/02/2024	Initial draft template	Antonio Sciarappa – LDO
Issue 2	14/03/2024	First version	All partners
Issue 3	27/03/2024	Second version	All partners
Issue 4	18/04/2024	Third version	All partners
Issue 5	24/04/2024	Review	Antonio Sciarappa – LDO
Issue 6	30/04/2024	Comments solved, deliverable cleaned	Antonio Sciarappa – LDO
Issue 7	10/05/2024	Ready for submission	Antonio Sciarappa - LDO

Table of Contents

Table of Contents 3
1 Introduction 4
1.1 Executive Summary 4
1.2 Definitions and Acronyms 4
2 Space Demonstrator 7
2.1 Overview of the Demonstrator 7
2.1.1 Multi-spectral image processing 7
2.1.2 Anomaly detection from telemetry data 9
2.2 High-level Architecture 10
2.2.1 Cores 11
2.2.2 Accelerators 11
2.2.3 Software 13
3 Automotive Demonstrator 16
3.1 Overview of the Demonstrator 16
3.2 High-level Architecture 19
3.2.1 Cores 24
3.2.2 Accelerators 24
3.2.3 Software 25
4 Smart Home Demonstrator 26
4.1 Overview of the Demonstrator 26
4.2 High-level Architecture 26
4.2.1 Cores 27
4.2.2 Accelerators 30
4.2.3 Software 30
5 Cellular IoT Demonstrator 33
5.1 Overview of the Demonstrator 33
5.2 High-level Architecture 33
5.2.2 Communication Subsystem 35
5.2.3 Software 36
5.3 Validation and Testing 36
6 Conclusions 38
7 References 39
8 Annex 40

1 Introduction

1.1 Executive Summary

This Deliverable reports the proposed architectures of the four Demonstrators (Space, Automotive, Smart Home, Cellular IoT) in four separate sections, as well as providing a more detailed description of the use case applications. A consolidated version of the Demonstrator requirements is instead provided in Deliverable D1.3, written at the same time. All partners involved in WP5 contributed to this Deliverable.

1.2 Definitions and Acronyms

Abbreviation	Description
AC	Alternating Current
AES	Advanced Encryption Standard
AI	Artificial Intelligence
ALU	Arithmetic and Logic Unit
AOT	Ahead-Of-Time
API	Application Programming Interface
AR	Auto Regressive
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuits
AXI	Advanced Extensible Interface
BLAS	Basic Linear Algebra Subprograms
BLDC	Brushless DC
CAE	Convolutional Autoencoders
cIoT	cellular IoT demonstrator
CLIC	Core Local Interrupt Controller
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DC	Direct Current
DDR	Double Data Rate
DFE	Digital Front-End
DL	Deep Learning
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EO	Earth Observation

Abbreviation	Description
FDIR	Fault Detection Isolation and Recovery
FOC	Field Oriented Control
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GEMM-Ops	General Matrix-Matrix Operations
HMAC	Hash Message Authentication Code
HPC	High-Performance Computing
HW	Hardware
IoT	Internet of Things
IP	Intellectual Property
IQ	In-phase and Quadrature
IR	Infrared
ISA	Instruction Set Architecture
KMAC	KECCAK Message Authentication Code
KEM	Key Encapsulation Mechanism
LCA	Loosely-Coupled Accelerator
LLC	Last Level Cache
LLVM	Low Level Virtual Machine
LSTM	Long-Short-Term-Memory
LTE	Long Term Evolution
ML	Machine Learning
MMU	Memory Management Unit
NAS	Neural Architecture Search
NN	Neural Network
OOL	Out-Of-Limits
OS	Operating System
PCA	Principal Component Analysis
PLIC	Platform Local Interrupt Controller
PMC	Performance Monitoring Counters
PMSM	Permanent Magnet Synchronous Motor
PQC	Post-Quantum Cryptography
PULP	Parallel Ultra Low Power
RAM	Random Access Memory
RF	Radio Frequency

Abbreviation	Description
RISC	Reduced Instruction Set Computer
RNN	Recurrent Neural Network
ROM	Read-Only Memory
RoT	Root of Trust
RTL	Register Transfer Logic
RVS	Rapita Verification Suite
SHA	Seccure Hashing Algorithm
SIMD	Single Instruction Multiple Data
SME	Smart Energy Management
SMGW	Smart Meter GateWay
SoC	System-on-Chip
SSH	Secure SHell
SW	Software
TCA	Tightly-Coupled Accelerator
TPE	Tensor Processing Engine
VLSU	Vector Load/Store Unit
VRF	Vector Register File
WCET	Worst-Case Execution Time
XREG	EXchange REgister

2 Space Demonstrator

2.1 Overview of the Demonstrator

Nowadays, due to the limited computational capabilities of modern satellites, the data collected in space is not, if not minimally, used directly on-board but is sent to Earth and then processed and consumed. Nevertheless, having a higher computational power in orbit would enable a whole series of new capabilities, from automated debris avoidance to real-time system monitoring.

The purpose of the Space Demonstrator is to evaluate the potentialities of RISC-V high performance cores with dedicated accelerators for intensive computations on satellites. In particular, two scenarios will be addressed:

- Multi-spectral image processing
- Anomaly detection from telemetry data

2.1.1 Multi-spectral image processing

One of the proposed use cases examines the use of satellite imagery for forest fire detection. Generally, performing processing on data from multispectral sensors is prohibitively expensive for onboard analysis. Indeed, datasets used for Earth observation (EO) are based on pre-processed imagery including at least basic operations such as orthorectification, co-registration and calibration, as well as filtering to reduce noise and distortion. Of course, each of these operations adds to the computational burden that makes it more difficult to apply Machine Learning (ML) / Deep Learning (DL) models that are usually trained on such processed images.

One possible solution involves simplifying the processing so that DL networks can be trained on this data. For this reason, the use case starts with a dataset whose processing is compatible with onboard operations along with the subsequent processing needed for forest fire reporting.

To overcome this problem, the demonstrator proposed here will use, as starting point, significantly rougher data that require only an equalization and a compression operated on-board to enable the processing of onboard-acquired data through a DL model optimized for efficient operation on a RISC-V based processor. Actually, the data that will be used for the demonstrator, after the equalization and the compression are sent to the ground station to produce metadata, including geographical information. However, for the scope of the demonstrator, this ancillary information is used only for the labelling part to locate the target events (wildfire) into the raw images. After the labelling and the training of the model, this second step including the downlink is not more required before the inference of the model. This dataset aims to promote the development of energy-efficient pre-processing algorithms and artificial intelligence models for applications aboard satellites. Despite being conceived for the detection of thermal hotspots events, the proposed approach is widely applicable for the design of any classification/detection tasks on Sentinel-2 raw data.

Dataset

The dataset identified for this purpose is THRawS (https://arxiv.org/abs/2305.11891). This dataset includes satellite data containing temperature hotspots, such as fires and volcanic eruptions, from around the world. These raw data are obtained after a lightweight “coarse spatial co-registration” and georeferencing. This process is fast but provides sufficiently accurate spatial registration to locate events on L0 data after detection on Level-1C (L1C) data. The dataset includes more than 100 samples, including fires, volcanic eruptions, and event-free volcanic areas, to enable hot event detection and general classification applications.

Model

The models intended to be used for satellite hotspot recognition are algorithms already widely used for hyper- and multispectral image processing. Among the most established are convolutional models, which analyze images using convolutional filters. A more appropriate variant for hyper-spectral image processing is given by models using 3D convolutions, which add to the spatial dimension the dimension related to the bands that make up the spectral hypercube.

Unlike traditional 2D convolutions, which operate on individual spectral bands separately, 3D convolutions can capture spectral and spatial features jointly, providing a more comprehensive representation of the hyperspectral data. By considering the entire spectral cube as a 3D volume, 3D convolutional layers can learn spatial and spectral patterns that are not easily detectable. The proposed Convolutional block comprises a 3D convolution layer with a kernel size of 3 × 3 × 3 followed by another 3D convolution layer with a kernel size of 3 × 1 × 1 and stride 2 × 1× 1 to reduce the spectral dimension. The convolutional block used in this architecture is shown in Figure 1 below. The number of filters of the first layer is set to a number n while for the second and third convolutional layers, this value increases by 3/4 of the previous value.

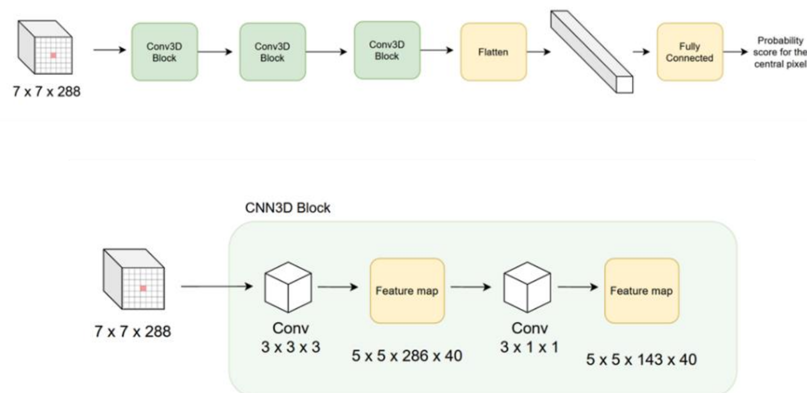


Figure 1: 3d convolutional layer block

Convolutional networks are not the only ones that can be used for forest fire detection. Fully connected networks can also possibly be explored even if the domain of the images is not what they are used to processing by their nature. This is because by their nature they would process each pixel without looking at the spatial context around it, which generally provides useful information to improve classification accuracy.

Models Parameters

The parameters of these models are the weights that effectively set the domain of discrimination between the event you want to identify and the rest of the zones that instead represent the uninteresting background. The more parameters a model has, the more complex the function that can increase its accuracy. This discussion, however, should not be unrelated to the dimension of the dataset, which currently has about fifty fires that can be used to drive and validate the model. However, each event contains a hundred pixels covering the forest parts affected by a fire. To this basic data, new data can be obtained with augmentation techniques to increase the generalization of the model. It is expected that a simple baseline could have a number of parameters in the tens of thousands but with the possibility of being able to increase this number to a few hundred thousand parameters.

Software

The software component focuses on the implementation of a deep learning model tailored for the unique constraints of onboard processing. Specifically, the model is designed to operate

seamlessly with limited computational resources while directly applying to Raw Sentinel-2 data, bypassing the conventional pre-processing step.

2.1.2 Anomaly detection from telemetry data

Another of the proposed use cases for the Space demonstrator regards Satellite Health monitoring and Telemetry Analysis, with a focus on anomaly detection. This use case falls inside the Fault Detection Isolation and Recovery (FDIR) domain, in particular in the Fault Identification part. In fact, the ML-based methods described in this section are used to detect anomalous behavior with respect to nominal spacecraft operation. The development of ML-based methods is indeed carried out with the objective to overcome limitations of traditional methods, which relies heavily on domain expert knowledge and the automation is usually limited to Out-Of-Limits (OOL) checks. A high-level architecture for raw signal to anomaly detection with ML methods is represented in Figure 2. TASI has accumulated years of experience in this area, and it is now focusing on consolidating these developments, as well as deploying them on HW. One key aspect is indeed to analyze telemetry data directly on-board and RISC-V processors fall into this scenario.

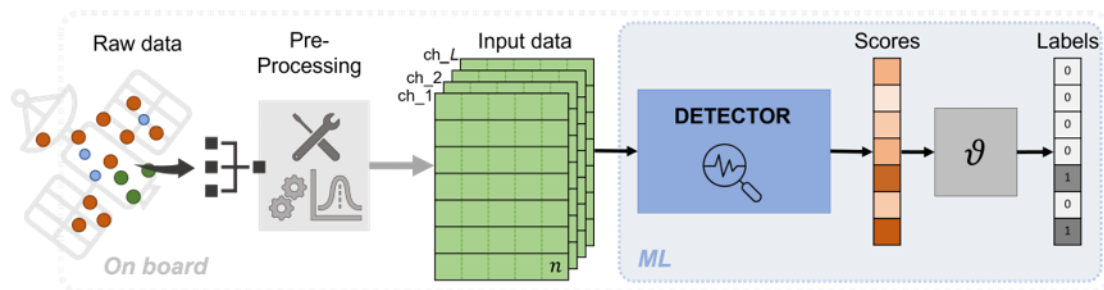


Figure 2: Signal processing and score computation chain for Anomaly detection applications from [1]

Dataset

The dataset used to train the methods discussed below is based on telemetry data from the on-board Attitude Control Subsystem of a satellite. The anomaly is actually simulated by faking a failure of the monitored components.

The preprocessing may differ slightly between different methods, but it usually includes the following tasks:

- Data polishing: only nominal data should reside in the dataset,
- Windowing: the ML models are fed with a frame of predefined length of the time series,
- Normalization: data is normalized between 0 and 1.

Models

For the purpose of this demonstrator, the methods considered are the following:

- Principal Component Analysis (PCA), that is a linear dimensionality reduction technique, which is indeed able to linearly transform the input into a reduced representation such that its principal components express the maximum amount of variance.
- Auto Regressive (AR), that is a stochastic method largely used to predict future values of a time series based on its previous values.
- Convolutional Autoencoders (CAE), a DL method featuring Encoder-Decoder architecture. The encoder is used to encode the input data into a latent space, with a reduced dimension, and the decoder task is to reconstruct the original data. The encoder and the decoder are usually designed to be symmetric and in this case are built using 1D convolutional layers, as the input data is composed by telemetry time series data.
- Long-Short-Term-Memory (LSTM), another DL method, in this case a Recurrent Neural Network (RNN), which is able to predict future samples of the time series.

An example for an Encoder-Decoder architecture for anomaly detection, valid for both PCA and CAE, is reported in Figure 3.

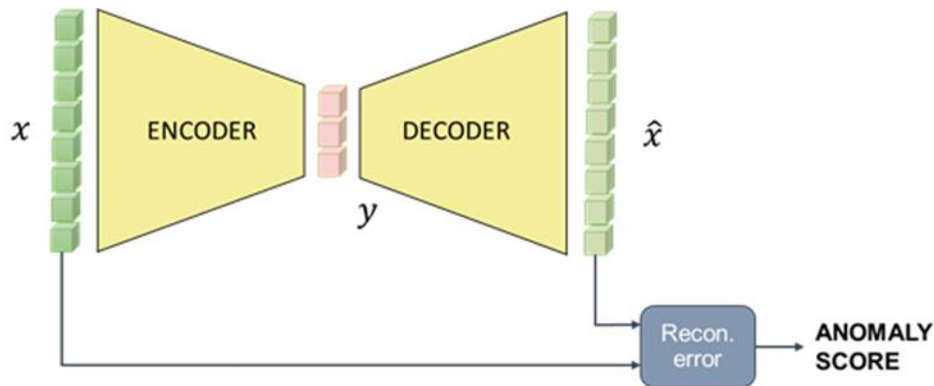


Figure 3: Encoder-Decoder architecture typical of Autoencoders, including the detail of the computation of an anomaly score based on the reconstruction accuracy

Software

All software implementation is focused on the ML and DL methods. All the methods are implemented in Python, in particular the PCA implementation is based on the one available on scikit-learn and the Neural Networks (NNs) (i.e. CAE and LSTM) are implemented using PyTorch. A low level (C) implementation is available, but it is worth noting that it was developed for a different Hardware (HW) architecture. The methods feature different characteristics, e.g. computational cost, memory footprint as well as scalability, and it is envisaged, especially for NNs, to exploit HW acceleration.

2.2 High-level Architecture

The proposed architecture is based on the open-source Cheshire template (https://github.com/pulp-platform/cheshire) with several additions and improvements. We envision a system divided in several domains, some of which are adapted from the Cheshire template (black), others are contributions in the ISOLDE project (blue):

- *Host domain*, composed of a host processor based on CVA6 + data/instruction caches, an L2 cache, a DMA controller, and a boot ROM;
- *Peripheral domain*, composed of peripherals such as JTAG, I2C, QSPI to connect with external devices;
- *System-level AXI crossbar,* which is used to connect all domains together;
- *Root-of-Trust (UNIBO)*, a RoT unit based on OpenTitan https://github.com/lowRISC/opentitan providing facilities for secure computing, such as secure boot, cryptographic primitives;
- *Accelerating cluster 0* (PULP + TPE), based on the PULP cluster https://github.com/pulp-platform/pulp_cluster integrating a *Tensor Processing Engine (UNIBO)* along with multiple DSPs based on the RISC-V instruction set architecture;
- *BIKE (POLIMI)* to provide post-quantum cryptography primitives support;
- *Parallel Computing Accelerator (POLITO)* to provide accelerated approximate computing capabilities.

Each domain can communicate with the Linux-capable CVA6 host through the AXI system-level crossbar. A scheme of the proposed architecture can be found in Figure 4.

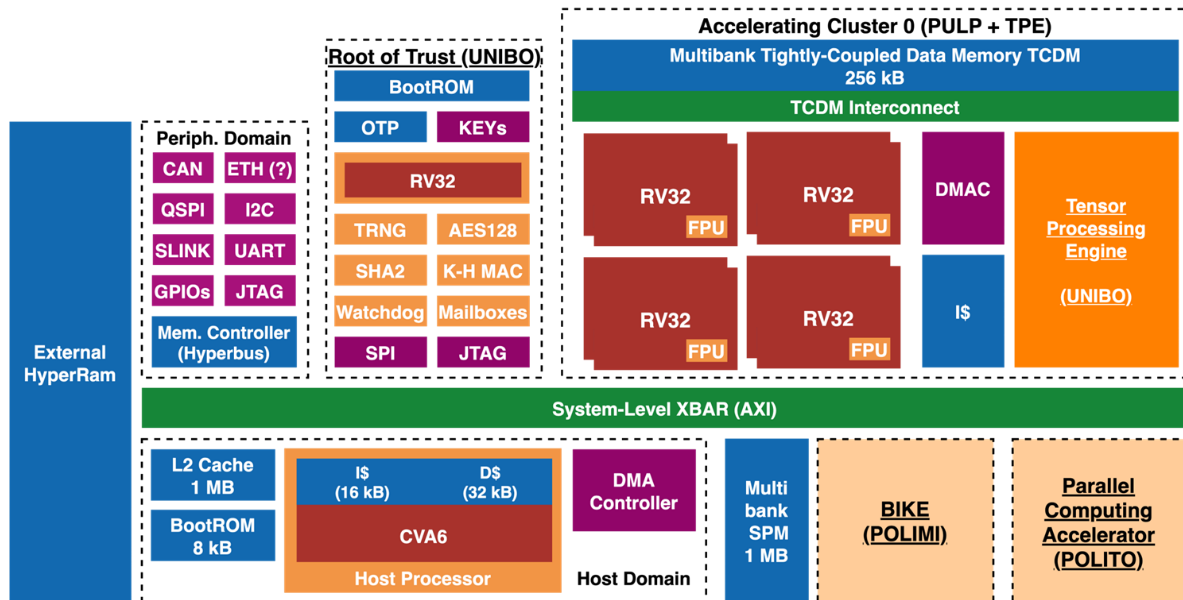


Figure 4: Space Demonstrator architecture proposal with details of cores and accelerators

2.2.1 Cores

Considering the applications characteristics and the requirements discussed in D1.1 and D1.3, the Space demonstrator will be based on a single 64-bit Linux-capable CVA6 host core. In addition, other items developed in WP2 will be considered for integration.

Context-Aware Bus and Core extensions (TRT)

CVA6 core and system bus will be extended with execution context information, in order to allow context-aware Performance Monitoring Counters. Such extensions will enhance current PMCs and enable more efficient safety verification and monitoring of the system, allowing the system to define and select only the events that need to be collected.

2.2.2 Accelerators

As far as accelerators are concerned, the following items developed in WP3 will be the main ones considered for integration in the Space demonstrator:

Context-Aware PMC and PMC Interface (TRT)

Advanced Performance Monitoring Counters will be developed for CVA6, with the capability of filtering the events based on a context (e.g. VM or process executing in the host core) on which the event was generated; in addition, an addressable interface will be developed to provide this access.

Parallel computing accelerator (POLITO)

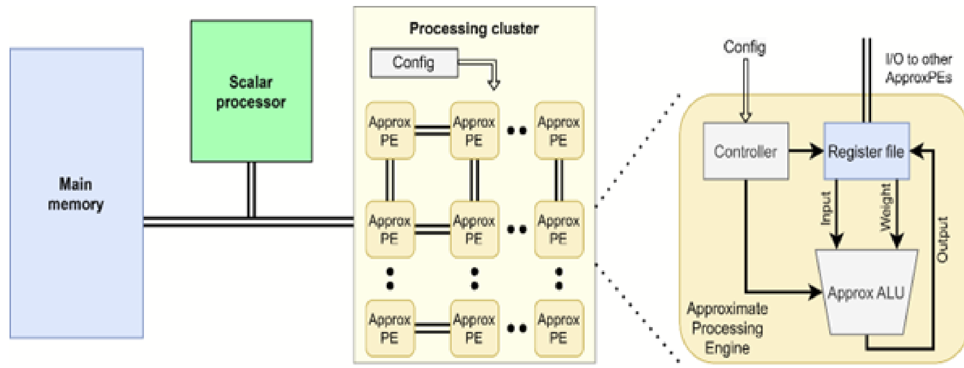
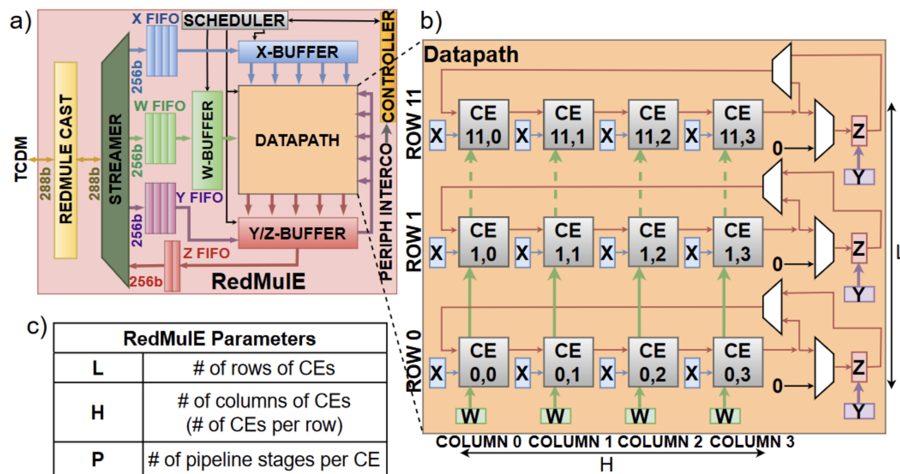


Figure 5: Architecture of the Parallel Computing Accelerator

As shown in Figure 5 above, the proposed parallel computing accelerator relies on an approximate processing cluster architecture. The cluster is made of a programmable number of approximate processing elements, each of which contains a register file and an Arithmetic and Logic Unit (ALU). The accelerator can be configured via different parameters. Part of these parameters (such as the maximum number of processing elements and the maximum precision) are configured at design time, while other parameters (such as the approximation level) can be set at the run-time. The accelerator is connected to the CVA6 system architecture through an AXI interface and it processes a subset of ALU operations in approximate mode.

Tensor Processing Engine (UNIBO)



RedMulE Parameters	
L	# of rows of CEs
H	# of columns of CEs (# of CEs per row)
P	# of pipeline stages per CE

Figure 6: Architecture of the Tensor Processing Engine

The Tensor Processing Engine (TPE) depicted in Figure 6 is a low-power specialized accelerator conceived for multi-precision floating-point General Matrix-Matrix Operations (GEMM-Ops) acceleration, supporting FP16, as well as hybrid FP8 formats, with {sign, exponent, mantissa}=({1,4,3}, {1,5,2}). The TPE is integrated within a PULP cluster for cooperative computation together with the cluster's DSPs.

Root-of-Trust Unit (UNIBO)

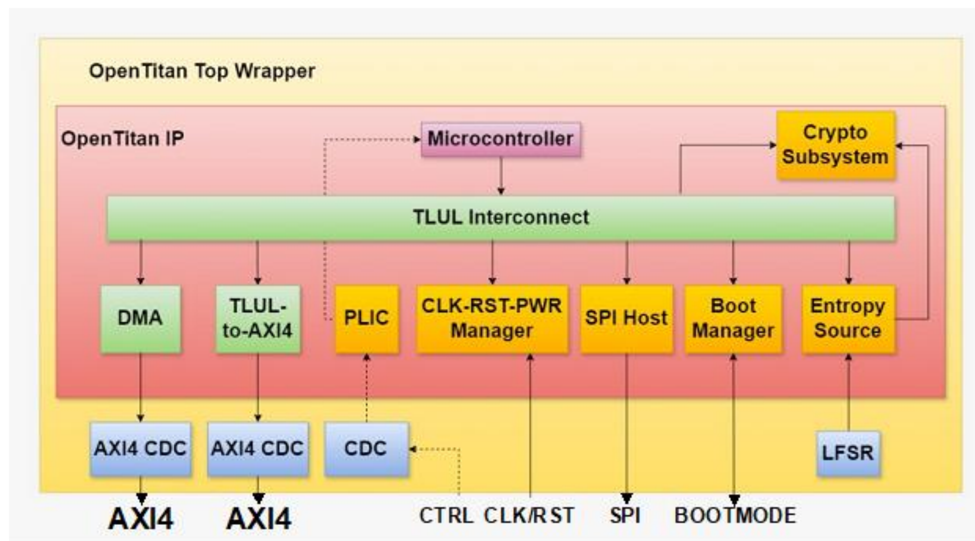


Figure 7: Architecture of the Root-of-Trust unit

The Root-of-Trust provided by UNIBO within ISOLDE is based on lowRISC's OpenTitan, the first open-source RISC-V based RoT design. It includes acceleration units for cryptographic hashing (SHA-256 and SHA-3), message authentication (HMAC, KMAC) and symmetric encryption (AES); a schematic representation of the unit is provided in Figure 7. UNIBO's RoT is meant to be a ready-to-integrate silicon IP able to act as a RoT.

BIKE accelerator (POLIMI)

This accelerator provides hardware support for the key generation, encapsulation, and decapsulation primitives of the key encapsulation mechanism BIKE, thus adding post-quantum cryptography capabilities to the system. The accelerator is connected to the CVA6 system architecture through an AXI interface to exchange commands, related to the execution of the various KEM primitives, and data, corresponding to the public-private keypairs, plaintexts, and ciphertexts used or produced by the three KEM primitives.

Additional accelerators that may be considered are the following ones:

Parallel Scratchpad Memory (IMT)

The parallel scratchpad memory employs a multibank implementation to provide high-speed access to data with regular dense access patterns (e.g., lines, columns, 2D matrices, diagonals). This may accelerate the execution of the memory intensive portions of the applications by reducing the data transfers to the off-chip memory. The Scratchpad Memory architecture is described in detail in Deliverable D3.1.

SIMD/Vector Accelerator (IMT)

The SIMD/Vector accelerator can exploit the available Data Level Parallelism in the application and process multiple data elements in parallel. The SIMD/Vector Accelerator architecture is described in detail in Deliverable D3.1.

2.2.3 Software

The following software tools developed in WP4 are planned to be used in the context of the Space demonstrator.

Optimized DNN Software Kernels (POLITO)

POLITO focused on gathering specifications from the space demonstrator and initiating the creation of kernels, aimed at enhancing coordination between the CVA6 cluster and the vector accelerator proposed in the main demonstrator architecture. These efforts are directly applied to the optimization of the execution of the end-to-end Deep Neural Networks (DNNs) for forest fire detection using satellite data. Enabling the on-board processing and hardware exploitation throughout kernel designing, the system effectively minimizes the need to transmit vast amounts of data back to Earth, significantly enhancing the efficiency and responsiveness of satellite operations.

DNN Optimization Toolchain (POLITO)

POLITO will exploit its advancements in Neural Architecture Search (NAS) to optimize the architecture of forest fire detection CNNs, aiming to reduce their size and complexity for efficient on-chip deployment. The integration of quantization within NAS further enables the conversion of the network to integer types, compatible with the hardware of the space demonstrator. The integration of this step is crucial for the space demonstrator to allow the hardware to run more complex AI models more efficiently, thereby reducing power consumption while improving the accuracy of the on-board detection, trying to match the ones that can be reached on HPC systems.

System-level Simulation of Extra-functional Properties (POLITO)

POLITO also contributes to the demonstrator by providing a SystemC-based simulation framework able to model (parts of) the complete system, focusing on non-functional properties and, in particular, on power consumption aspects. The framework supports the simulation of different sub-systems including sensors, energy storage (batteries) and energy sources (e.g. PV panels). In the initial part of the project, the activities of this work item focused on identifying ways to integrate the SystemC-based simulation with software execution running on CVA6.

Optimizing Power/Performance: Best CVA6 Configurations (Silvaco)

Silvaco will optimize algorithm power/performance trade-off on the CVA6 architecture. Focusing on benchmarks such as matmul, we'll identify the best configurations within Thales constraints parameters ranges, balancing power consumption and computational efficiency. Through simulation and data generation based on RTL and GL simulations, we aim to understand CVA6 behavior and develop machine learning models for predicting power and performance. The anticipated outcome is a suite of best-fit configurations for the CVA6 processor, which are expected to enable efficient processing for space applications while remaining within the energy constraints of such missions. These configurations will be derived from simulations, with the understanding that implementation on an FPGA board falls outside the scope of the limited efforts Silvaco has dedicated to the demonstrator.

Inference timing analysis (Rapita Systems)

Rapita Systems will provide an identification of interference channels and a study of main critical configuration settings having an impact on timing interference, based on the sharing of memory resources (memory, buses) between the CVA6 core and accelerators in the system on chip. Support on RVS RapiTime tool for the actual target for measurement of relevant hardware metrics and Worst-Case Execution Time (WCET) in contention scenarios will be provided. Furthermore, support for running this tool over XNG hypervisor is expected.

XNG Hypervisor virtualization support (FENTISS)

FENTISS contributes to the demonstrator by supporting the evaluation of the XtratuM Next Generation (XNG) hypervisor over the CVA-6 core in the context of mixed critical applications,

e.g. situations with one critical application (e.g. telemetry / telecommand) and one non-critical application (e.g. the AI / ML models developed during the project) running together on different partitions; in such cases, the non-critical application will be using virtualized accelerators. The possibility of having multiple applications running on a partitioned accelerator may be evaluated at a later stage, depending on the behavior of the applications developed and the complexity of the accelerators themselves.

PikeOS for CVA6 (SYSGO)

Based on previous experience with other RISC-V cores, SYSGO will work towards providing CVA6 support for PikeOS. Based on its availability, the functionalities and capabilities of PikeOS on CVA6 will tentatively be evaluated in the context of the Space demonstrator.

Compiler Support for Approximate Computing (POLIMI)

POLIMI will apply precision tuning to the space demonstrator application, where appropriate, to achieve better latency/energy efficiency and reduce resource usage on the target architecture.

The compiler will target the general-purpose accelerator cluster, although we will also explore the feasibility of targeting dedicated components – this depends on the code generation pipelines employed for the dedicated accelerators and their compatibility with the LLVM compiler system upon which POLIMI's precision tuning components are deployed.

Compiler Support for WCET estimation and optimization (POLIMI)

POLIMI will apply novel techniques inside the LLVM compiler to estimate the WCET during the compilation flow and properly exploit the compiler optimizations to reduce the WCET of the application. The resulting tool will be tentatively exploited for the RISC-V architecture and the software applications of the space demonstrator.

3 Automotive Demonstrator

3.1 Overview of the Demonstrator

Eye detection (see example in Figure 8) is an essential feature for driver monitoring systems acting as a base functionality for other algorithms like attention or drowsiness detection.

Figure 8: An example of eye detection

Multiple methods for eye detection exist. The machine learning based methods involve a manual labeling process in order to generate training and testing datasets. This use case presents an eye detection algorithm based on convolutional neural networks trained using automatically generated ground truth data and proves that we can train very good machine learning models using automatically generated labels. Such an approach reduces the effort needed for manual labeling and data preprocessing and it is applicable in image processing.

The algorithm and the relative processing engines will be implemented on an FPGA based setup and will be validated using laboratory and real in-car environment setup.

Algorithms will be fed with images from cameras with IR image sensors and illumination and the results can be tracked /analyzed on a PC, Intel Core i7-10700, 16 GB, Intel UHD graphics 630, 512 GB, M.2 PCIe as depicted in Figure 9 and Figure 10.

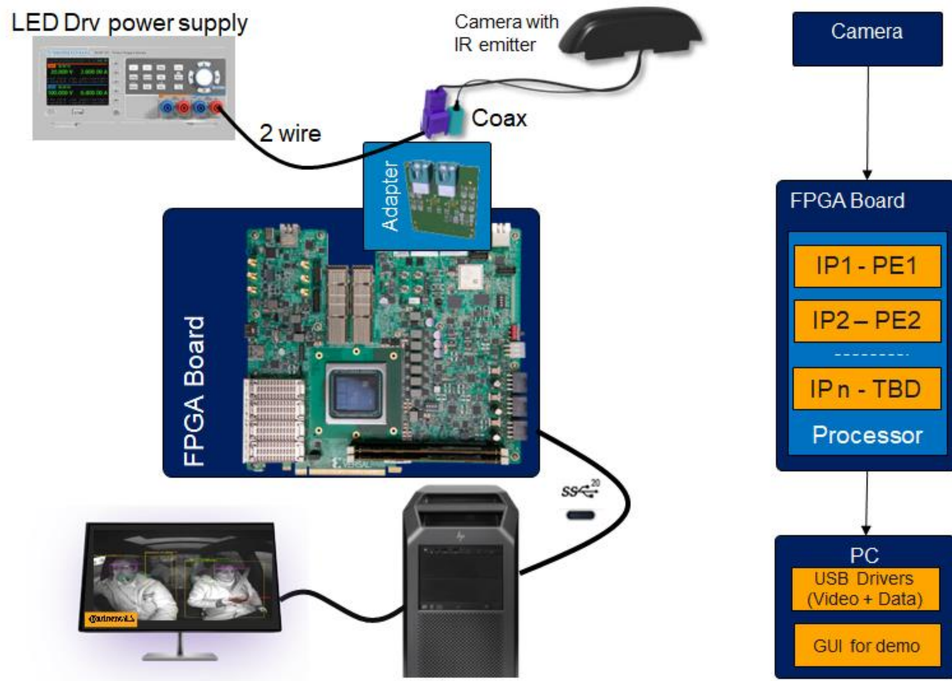


Figure 9: Block diagram of the demonstrator setup

Figure 10: Possible position of the camera in a car

In addition, Electronic Motor control is an essential feature that is demonstrated in Isolde. As electronic motors consume a substantial amount of energy worldwide, this is also a contribution to worldwide CO2 reduction.

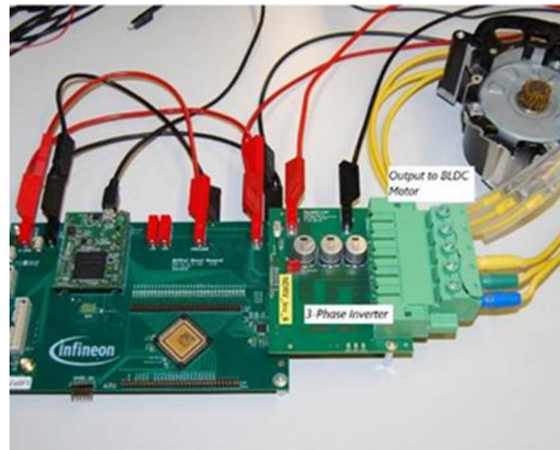


Figure 11: Potential Hardware Demonstrator

The deliverable is an ASIC (or as backup an FPGA) interfacing the motor electronic. The demonstrator is from automotive domain and can be used on a slightly modified way in Industrial applications. Figure 11 shows a potential setup of the demonstrator.

As the demonstrator is close to a non RISC-V product, there is a realistic chance that the demonstrator will be enhanced to a motor control product. A second benefit is that hand-written software exists and efforts for migrating software can be analysed. The RISC-V has special instructions that boost the performance for DSP and AI applications in the range of High-Performance-Computers.

There are different kinds of motors, which can be classified by the way they are constructed. A BLDC (Brushless DC) motor is an electric motor, which uses a permanent magnet in the rotor and an electronic controller to control the electric field made by the stator. In contrast to traditional DC motors, BLDC motors have no brushes, and the commutation is done electronically i.e. there is no mechanical commutation connection between rotor and commutator. Thus, BLDC motors are more reliable, and are often used when high efficiency, long lifetime, and a precise speed control are needed.

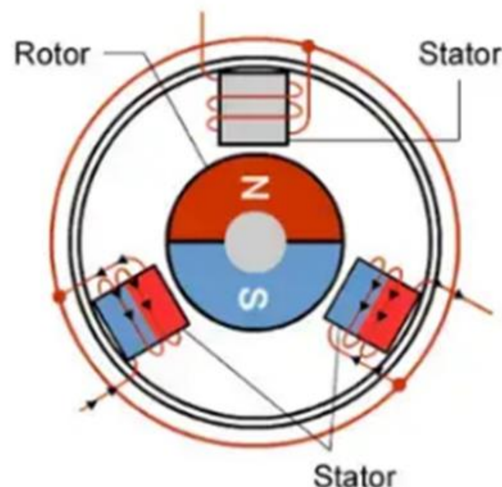


Figure 12: BLDC motor

Figure 12 shows the structure of a BLDC motor. The stators, consisting of a series of coils in a circular pattern, are situated on the walls of the motor and a permanent magnet, which serves as a rotor, is in the middle. The control is done electronically with an interaction of stator and rotor. A magnetic field is created by applying electric current to the stator coils. This interacts with the permanent magnet causing it to rotate. Timing and intensity of the current to the stator

coils are managed by the electronic controller. It is driven so that the magnetic field is always aligned with the permanent magnets in the rotor. This leads to an efficient and smooth rotation. Depending on the number of windings, a BLDC motor can be 1-phased, 2-phased or 3-phased.

Field Oriented Control (FOC) is a popular control algorithm for a 3-phase motor. The GenerIoT automatically built software should also be applied for this use case.

It is mostly used for PMSM (Permanent Magnet Synchronous Motor) and BLDC motors. For this control technique the Clarke-Park Transformation is used to transform the 3-phase AC voltage *U* and current *I*-signals into a two-phase reference frame. This decouples the magnetic flux and torque components of the motor and allows these to be controlled independently. This leads to better control of the torque response.

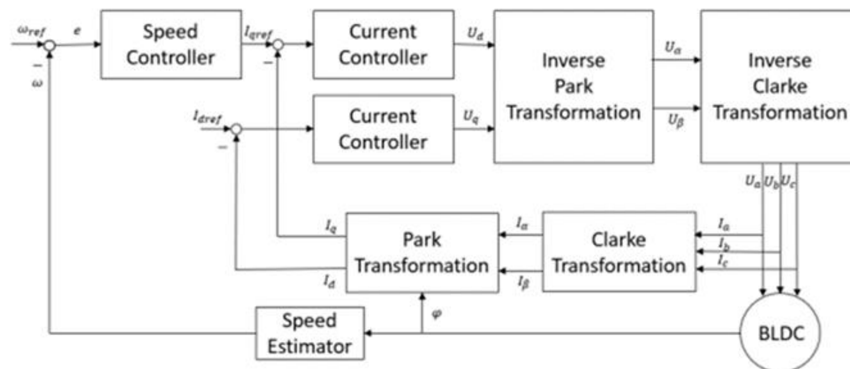


Figure 13: FOC block-diagram

Figure 13 shows the structure of the FOC, which consists of mainly three parts, the control, the transformations and the mechanical part including the BLDC motor. From the voltage controlling the BLDC motor U_a, U_b, U_c the currents I_a, I_b and I_c are calculated, which are then used as the input of the Clarke-Transformation. This transformation simplifies the system by converting the three-phase current into a two-phase current I_α and I_β.

These signals with addition of the motor angle φ are the input of the Park-Transformation, which rotates the received two-phase current into a stationary reference which are called direct current I_d and quadrature current I_q and thus separates them into the components that represent the torque and the magnetic field. The values of these signals build a difference with the reference signals I_{qref} and I_{dref} and are fed to the current controllers. I_{qref} is determined by the speed controller which contains a PI controller from the error e between the reference speed ω_{ref} and the actual speed ω. I_{dref} is often 0. The two current controllers output the direct voltage Ud and the quadratic voltage U_q, which is then transformed back to a three-phase signal with the Inverse Park-Transformation and the Inverse Clarke-Transformation. The immediate control of the stators is done in hardware thus the control algorithm needs to be executed in 100 Hz – 10 khz range, depending on the application.

3.2 High-level Architecture

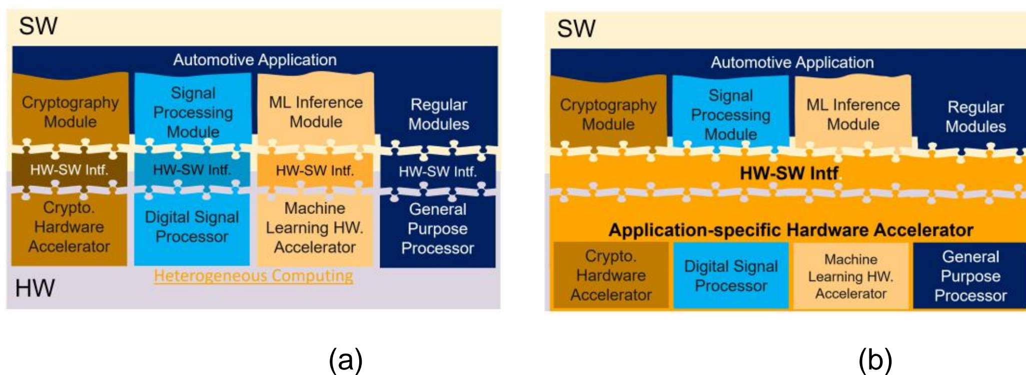
Automotive applications can be decomposed in:

- Compute-intensive modules: modulate flow of data, lots of number crunching and few decisions -e.g., machine learning inference, digital signal processing, cryptography.
- Regular modules: modulate flow of instructions, little number crunching and a lot of decisions.

In order to improve performance, compute-intensive module should be executed on a dedicated hardware accelerator. From this perspective, there are (at least) two options for hardware acceleration: hardware units that autonomously execute entire computational sub-graphs and instruction acceleration units, sometimes referred to as coprocessors, that take over complex instructions and thus are directly sequenced by the core instruction stream. Coprocessors imply less communication overhead, yet they can be efficiently exploited only within instruction set architectures (ISA) that allow extensions dedicated to particular computation domains [2]. In general, each selected hardware accelerator comes with its own hardware software interface (HW-SW Intf.) - see Figure 14. This approach leads to a heterogeneous hardware setup (General Purpose Processors, hardware accelerators, network to interconnect them) which can be a challenge to be programmed, due to the fragmented HW-SW Intf. A hardware Façade build on top of this heterogeneous accelerators set can make the programmer's life less challenging. Benefits of this approach:

- SW First! - hardware is customized for the application needs (i.e. the selected hardware accelerator(s) shall match the required performance/watt)
- A single hardware - software interface shall ease the programming of the system.
- Synergies at hardware level can be acquired, enforcing better performance/watt.

The above is known as Application-Specific Instruction-set Processor delivers high performance and energy efficiency by tailoring a processor architecture and instruction set to the specific requirements of a particular application domain.



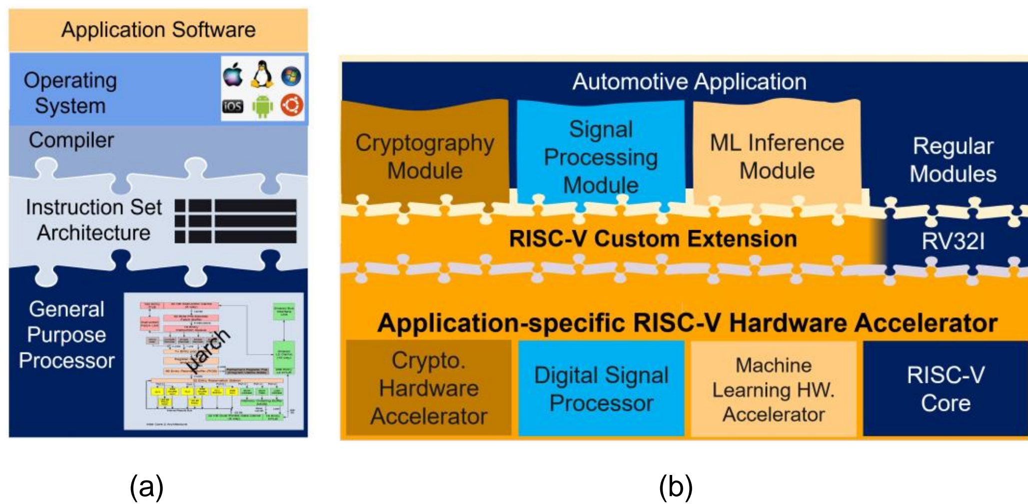
(a) (b)

Figure 14: Heterogeneous computing: (a) Fragmented HW-SW interface (b) Unified HW-SW interface

The HW-SW Interface of a computing system is known as Instruction Set Architecture (ISA) Instruction Set Architecture (ISA) defines:

- Instruction set and instruction encoding.
- Architectural state (register set, control & status register, etc.)
- Memory specification: addressing, alignment.
- Virtual Memory Architecture, etc.

The Microarchitecture (µarch) is the implementation of the instruction set architecture (ISA) in a particular general-purpose processor. A compiler will translate the high-level language into a flow of instructions defined by the ISA. An ISA is essentially a compiler target.



(a) (b)

Figure 15: Instruction Set Architecture: (a) SW stack (b) AIDA

CAR introduces AIDA - Application-specific RISC-V Hardware Accelerator with the following design goals:

- Performance, measured using throughput (operations per second) and latency (time to complete a task)
- Flexibility, the degree to which the accelerator can cope with variations of the application

The hardware Façade in this particular case, will be RISC-V RV32I with custom extensions. The custom extensions are tailored to make the hardware accelerator(s) programmable from the RISC-V software ecosystem. To accommodate the ML model complexity, a ML compiler shall translate the model's execution graph into a flow of RISC-V like instructions. Software tool-chain will be able to translate ONNX machine learning models/C++ applications into binary code to be executed by AIDA Tool-chain development effort will be concentrated in:

- ONNX Front-end
- llc RISC-V backend

while leveraging all the optimizations already available in LLVM framework.

Accelerator Coupling

[3] evaluated the following design options when coupling hardware accelerators:

- Tightly-Coupled Accelerators (TCAs) - The core shares key resources with the hardware accelerators (register file, memory-management unit (MMU) and L1 data cache), and thus stalls until the accelerator completes execution.
- Loosely-Coupled Accelerators (LCAs) - hardware accelerators are located outside the CPU core.

[3] concluded that LCAs offer better performance due to the possibility to use private memory blocks tailored to the accelerator needs. The private memory blocks (scratchpad memory) offer better data locality and therefore LCAs have an enhanced throughput over the TCAs. From the software perspective, memory mapped LCAs don't require to modify the CPU's ISA. The core can via load/store instructions configure and trigger the execution of a complete applica-

tion kernel, for instance a Fast-Fourier-Transform or a convolution. In the following, it is assumed that there is a hardware accelerator which implements convolution according to the description from [4].

Loosely Coupled Accelerator – Convolution

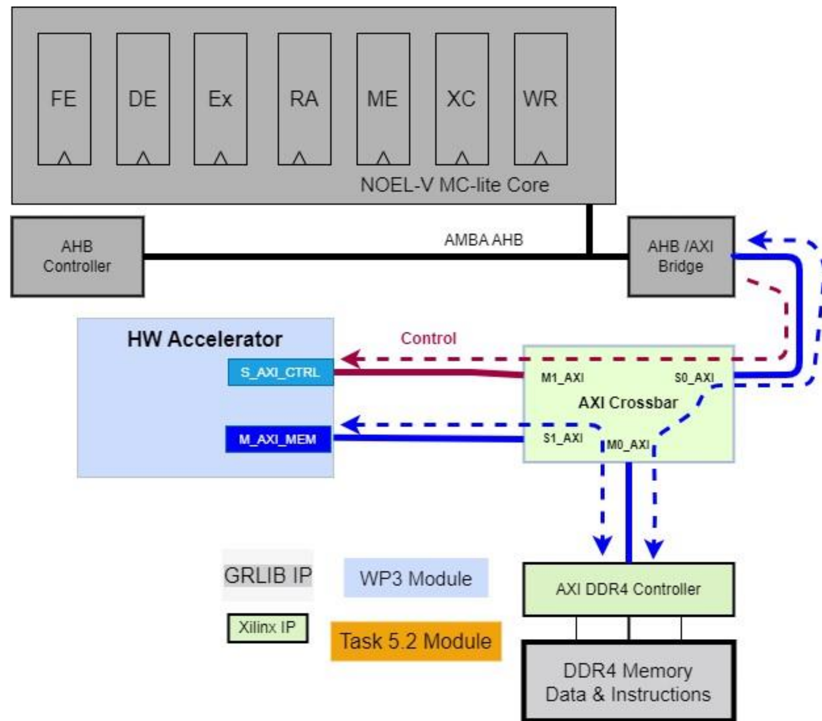


Figure 16: Loosely Coupled Accelerator on AXI bus

In the C++ application, it is assumed that S_AXI_CTRL port is mapped at 0x8000000.

From the software perspective, a simple C/C++ application, conv2d_api.cpp, is listed in the Annex.

Invoking the compiler,

```
clang -cc1 -S

-triple riscv32-unknown-elf -target-feature +v

-target-abi ilp32d -O3 -o conv2d_api.cpp
```

will generate a flow of 25 instructions, also available in Annex.

Tight Coupled Accelerator – Convolution

A coprocessor [5] shall support the following instruction types:

- load
- store
- processing instructions

The load and store instructions enable information to pass between:

- core file registers and coprocessor file registers

- memory and coprocessor

Any hardware accelerator which can be loosely-coupled to the core, can be "converted" into a coprocessor. From the software perspective, the core's ISA needs to be modified. To access the hardware, a build-in function was added to clang compiler, *__builtin_onnx_conv2df32*, see Annex for details. Invoking the compiler,

```
clang -cc1 -S

-triple riscv32-unknown-elf -target-feature +v

-target-abi ilp32d -O3 -o conv2d.cpp
```

will generate a flow of 13 instructions, 52 percent lesser than the LCA variant.

Future work

Depending on the hardware accelerator configuration complexity, it can be that integrating it as a coprocessor reduces the number of instructions. In the example above, the number of instructions (for equivalent C++ applications) was reduced from 25 to 13 instructions. Nevertheless, the tight coupling complicates the hardware integration. On the other hand, the RISC-V core shall offer just a thin hardware layer on top of the accelerator, therefore the core pipeline can be a simplified one. An option worth investigating would be to have a parallel pipeline for the custom RISC-V instruction.

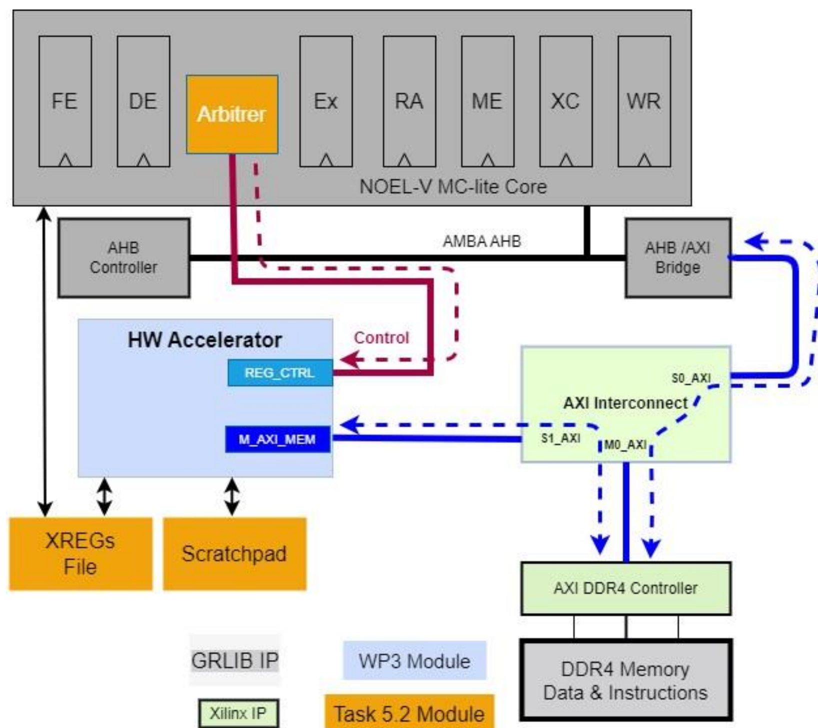


Figure 17: Tight Coupled Accelerator

CV-X-IF can be a candidate for the control interface.

In ISOLDE, we would like to integrate the hardware accelerator and the RISC-V core as a" MOLEN Polymorphic Processor". In ISOLDE the "reconfigurable processor" part is the hardware accelerator which is configurable but not reprogrammable (i.e. accelerator's structure is

frozen). As such, in the polymorphic ISA, the set instructions are dropped completely, and the exchange registers (XREGs) shall keep configuration and input/out parameters of the hardware accelerator. The “Arbiter” from [6] Fig.16 can be viewed as an additional stage in the RISC-V pipeline. Based on observation from [3] scratchpad memory should be added to increase the throughput.

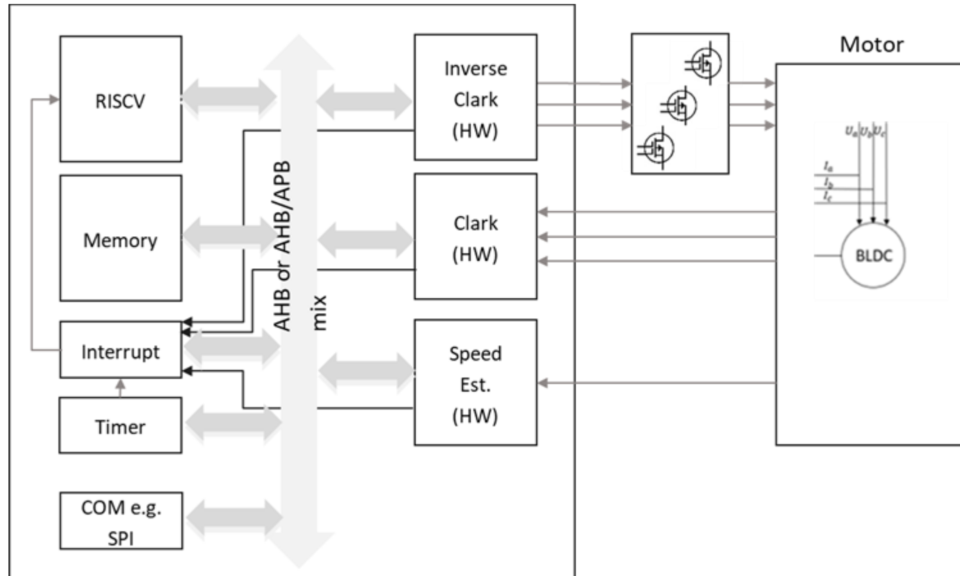


Figure 18: SoC block-diagram implementing Figure 13 FOC block diagram

Figure 18 shows high level block diagram for implementing the FOC.

The SoC will interface with the environment via SPI and will have a debug interface (not shown). The processor of the embedded CPU subsystem shall be a 32bit core. A bus bridge shall bridge different peripheral and memory accesses.

The preferred FPGA board for the integration of the demonstrator the Xilinx Versal VMK180 (but may be also an ARTY) for the following reasons:

IFX is experienced using Xilinx FPGA boards and Vivado toolchain. Prototyping behavior with an FPGA is an advanced methodology to be applied in the demonstrator.

For similar reasons the ASIC shall be implemented in an Infineon 130nm technology.

3.2.1 Cores

In the HW Facade we aim to modify the core pipeline, therefore we selected core will probably be NOEL-V.

Stretch goal: benchmark the several HW Facade implementations (listed in priority order):

- NOEL-V
- CVA6
- In-house HLS RV32I core (RISC-V core implemented in C++)
- In-house generated RV32IX Core

3.2.2 Accelerators

We will analyze as many as possible the use of such accelerators provided by the following partners:

- FotoNation (Tobii): AI/ML accelerator IP.
- UNIBO: Tensor Processing Unit.

- POLITO: Hardware accelerators for parallel processing.
- TUI: vector/SIMD units.
- SAL: Event-Based / Sparse convolution accelerator.
- SAL: Resource efficient PQC.
- IMT: SIMD/Vector Accelerator
- IAI: Closely coupled AI inference accelerator with Intrinsic support
- OFFIS: Time Contract Monitoring Co-Processor (TCCP)

3.2.3 Software

We will modify the following compilers:

Procedural compiler: The LLVM Compiler Infrastructure, gcc

ML compiler: onnx-mlir, tensor flow light micro (or TVM)

4 Smart Home Demonstrator

4.1 Overview of the Demonstrator

As described in D1.1, the goal is to show the usability of RISC-V application platforms for an SME in the energy sector. Figure 19 describes the setting for smart home energy management demonstrator:

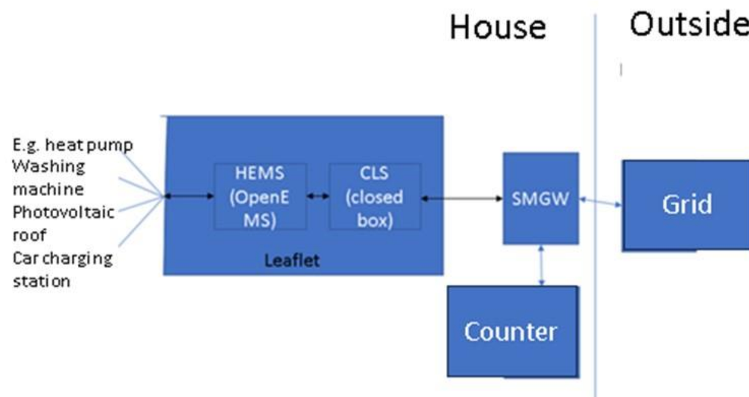


Figure 19: Setup of the Demonstrator

The application is an end consumer house network in which various electrical producers (e.g., PV on the roof and/or balcony module, wind turbine, etc.) and consumers (e.g., heat pump, heating rod, charging column for electric car, washing machine, light, etc.) communicate with the Leaflet device (see the Figure above). The goal is to optimize the local use of the generated energy using prognosis, e.g., the charging column battery, operation of the heat pump, are operated when electricity is provided from your own PV system. The decisions / optimizations are made here by a HEMS (Home Energy Management System), in our case based on OpenEMS (https://openems.io/), which is located on the Leaflet device developed by Consolinno. Since it is not cost-effective to strive for complete self -sufficiency in the event of a home network, there is still a connection to the public power grid via the supply system operator. In detail, the house network is separated from the supply system operator by a smart meter gateway (SMGW), which also accesses the counter. The SMGW communicates (e.g., via IEC 61850 protocol) with that of the Leaflet device, which also has a CLS unit (Controllable Local System).

The HEMS unit in the figure operates on the knowledge of all connected devices in the system, while the CLS unit only must know the aggregated value of the generation and consumption in order to communicate, whether fed or consumed, and in what amount this happens.

4.2 High-level Architecture

As described in D1.1 and D1.3, we target three implementation stacks, one on i.MX8 (ARM), two on RISC-V CVA-6, preferably multi-core. Concerning the two RISC-V CVA-6 stacks, one is a pure open source demonstrator that can be made fully available (e.g., on Github) as sample payload and the other CVA-6 stack is a demonstrator using a special kind of small codesize hypervisor (separation kernel, https://en.wikipedia.org/wiki/Separation_kernel), which is good at providing strong isolation of execution environments, and controlled execution between them. The separation kernel provided in this use case is called PikeOS (https://www.pikeos.com), and amongst others, has been evaluated against the security standard Common Criteria EAL5 (https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/Betriebssysteme/1146.html). The three targeted implementation stacks are shown in Figure 20.

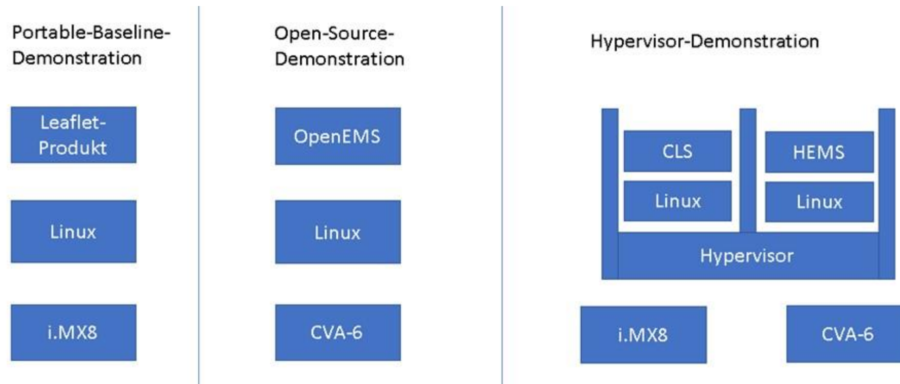


Figure 20: Targeted implementations of the Demonstrator

4.2.1 Cores

For the open-source demonstrator, we break it down to a core-agnostic RISC-V multicore qemu demonstration, and a demonstration on a single-core CVA6 demonstration (current version taken from CVA-6 git in September 2023).

For the portable-baseline architecture, the demonstration uses ARM i.MX8 processors.

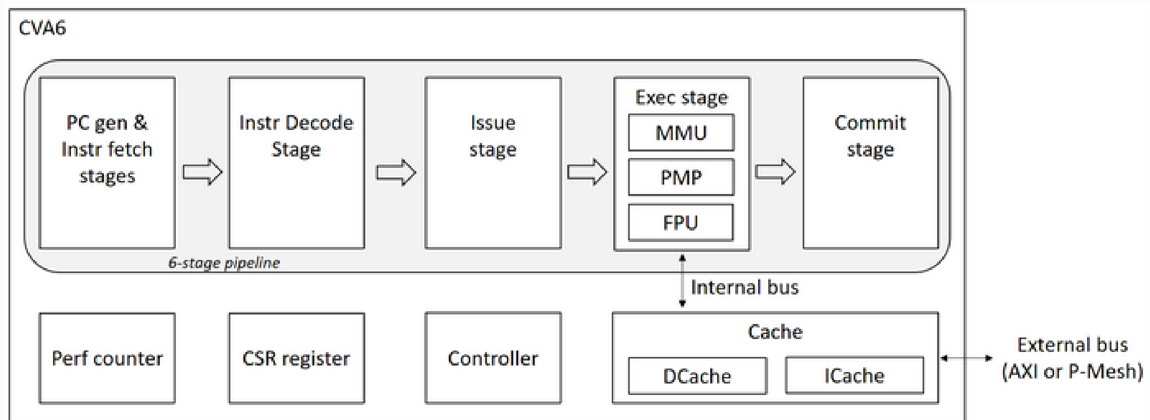
For the hypervisor demonstration it is again planned to use RISC-V CVA6.

CVA6 Cores

The scope of the default CVA-6 IP is described in https://cva6.readthedocs.io/en/latest/01_cva6_user/Introduction.html#scope-of-the-ip and is depicted in Figure 21, taken from the link indicated.

Scope of the IP

The **scope of the IP** refers the subsystem that is documented here.



As displayed in the picture above, the IP comprises:

- The CVA6 core;
- L1 write-through cache;
- Optional FPU;
- Optional MMU;
- Optional PMP;
- CSR;
- Performance counters;
- AXI interface;
- Interface with the P-Mesh coherence system of OpenPiton;
- CV-X-IF coprocessor interface (not shown).

Figure 21: Scope of the CVA6 IP

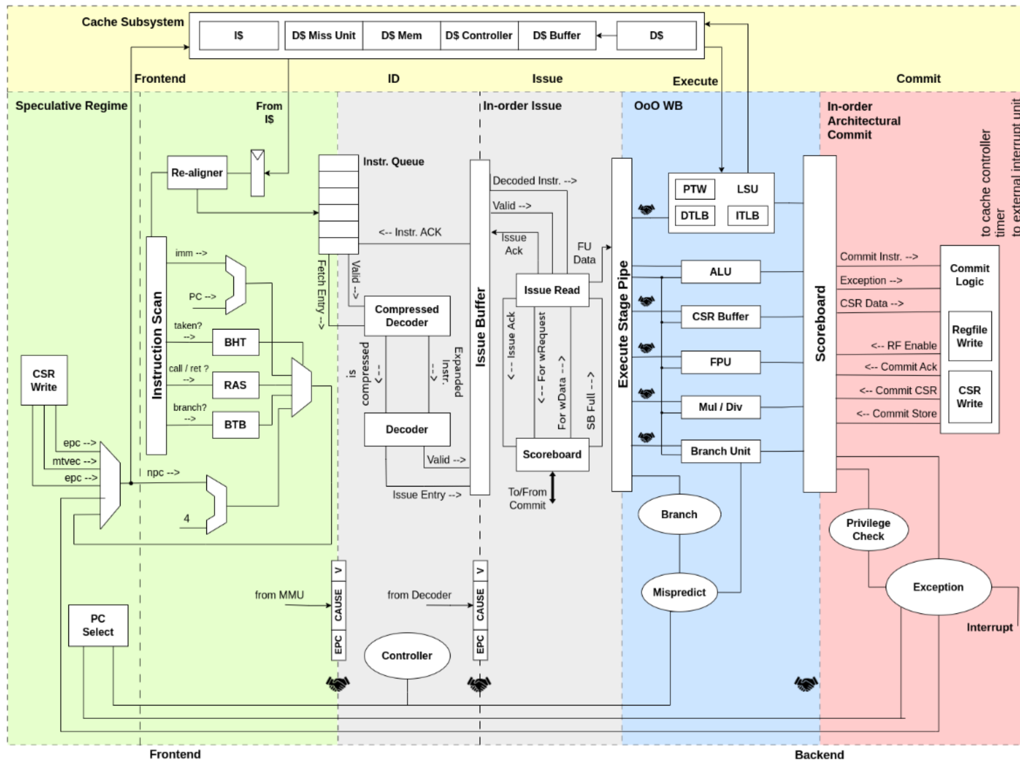


Figure 22: Block diagram of the used CVA6 IP (from https://github.com/openhwgroup/cva6)

ARM i.MX8 SoC

For the intentionally non-RISC-V baseline demonstrator of the energy management system we use an ARM i.MX8 processor.

i.MX 8 Processors

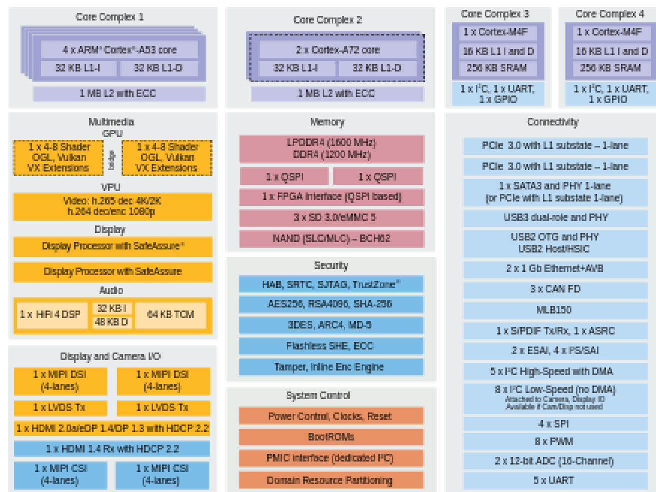


Figure 23: ARM i.MX8 block diagram as in https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8-family-arm-cortex-a53-cortex-a72-virtualization-vision-3d-graphics-4k-video:i.MX8

4.2.2 Accelerators

If available and if the integration is feasible, we want to make use of AI acceleration. This is an option, not a hard requirement.

4.2.3 Software

The software stack for the three sub-demonstrations is depicted in Figure 24.

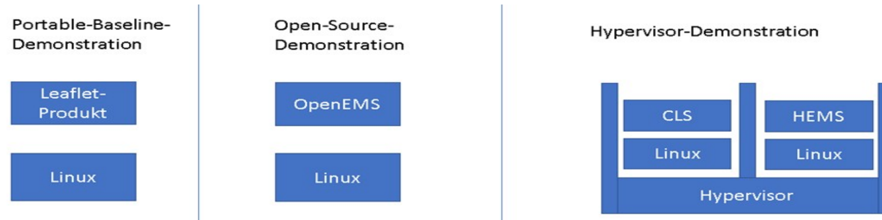


Figure 24: Software stack for the three sub-demonstrators

In general, we use a stack consisting of Linux, Java, OpenEMS as well as a version with virtualization consisting of PikeOS, Linux, Java, and OpenEMS.

For the open-source demonstrator in particular we have planned a configuration, which consists of the Debian packages for a base system, and OpenJDK-22 and its dependencies (e.g. libraries, packages beginning with “lib:”). A detailed list is given below:

adduser, adwaita-icon-theme, alsa-topology-conf, alsa-ucm-conf, apt, apt-utils, at-spi2-common, at-spi2-core, base-files, base-passwd, bash, bsdutils, ca-certificates, ca-certificates-java, coreutils, cpio, cron, cron-daemon-common, dash, dbus, dbus-bin, dbus-daemon, dbus-session-bus-common, dbus-system-bus-common, dbus-user-session, dconf-gsettings-backend:riscv64, dconf-service, debconf, debconf-i18n, debian-archive-keyring, debian-ports-archive-keyring, debianutils, diffutils, dmidecode, dmsetup, dpkg, e2fsprogs, fdisk, findutils, fontconfig, fontconfig-config, fonts-dejavu-core, fonts-dejavu-extra, fonts-dejavu-mono, gcc-13-base:riscv64, gpgv, grep, gsettings-desktop-schemas, gtk-update-icon-cache, gzip, hicolor-icon-theme, hostname, ifupdown, init, init-system-helpers, initramfs-tools, initramfs-tools-core, iproute2, iputils-ping, java-common, klibc-utils, kmod, less, libacl1:riscv64, libapparmor1:riscv64, libapt-pkg6.0:riscv64, libargon2-1:riscv64, libasound2:riscv64, libasound2-data, libatk-bridge2.0-0:riscv64, libatk-wrapper-java, libatk-wrapper-java-jni:riscv64, libatk1.0-0:riscv64, libatomic1:riscv64, libatspi2.0-0:riscv64, libattr1:riscv64, libaudit-common, libaudit1:riscv64, libavahi-client3:riscv64, libavahi-common-data:riscv64, libavahi-common3:riscv64, libblkid1:riscv64, libbpf1:riscv64, libbrotli1:riscv64, libbsd0:riscv64, libbz2-1.0:riscv64, libc-bin, libc6:riscv64, libcairo-gobject2:riscv64, libcairo2:riscv64, libcap-ng0:riscv64, libcap2:riscv64, libcap2-bin, libcbor0.10:riscv64, libcom-err2:riscv64, libcrypt1:riscv64, libcryptsetup12:riscv64, libcups2:riscv64, libdatrie1:riscv64, libdb5.3:riscv64, libdbus-1-3:riscv64, libdconf1:riscv64, libdebconfclient0:riscv64, libdeflate0:riscv64, libdevmapper1.02.1:riscv64, libdrm-amdgpu1:riscv64, libdrm-common, libdrm-nouveau2:riscv64, libdrm-radeon1:riscv64, libdrm2:riscv64, libedit2:riscv64, libelf1:riscv64, libexpat1:riscv64, libext2fs2:riscv64, libfdisk1:riscv64, libffi8:riscv64, libfido2-1:riscv64, libfile-find-rule-perl, libfontconfig1:riscv64, libfreetype6:riscv64, libfribidi0:riscv64, libgail-common:riscv64, libgail18:riscv64, libgcc-s1:riscv64, libgcrypt20:riscv64, libgdbm-compat4:riscv64, libgdbm6:riscv64, libgdk-pixbuf-2.0-0:riscv64, libgdk-pixbuf2.0-bin, libgdk-pixbuf2.0-common, libgif7:riscv64, libgl1:riscv64, libgl1-mesa-dri:riscv64, libglapi-mesa:riscv64, libglib2.0-0:riscv64, libglib2.0-data, libglvnd0:riscv64, libglx-mesa0:riscv64, libglx0:riscv64, libgmp10:riscv64, libgnutls30:riscv64, libgpg-error0:riscv64, libgraphite2-3:riscv64, libgssapi-krb5-2:riscv64, libgtk2.0-0:riscv64, libgtk2.0-bin, libgtk2.0-common, libharfbuzz0b:riscv64, libhogweed6:riscv64, libice-dev:riscv64, libice6:riscv64, libicu72:riscv64, libidn2-0:riscv64, libjansson4:riscv64, libjbig0:riscv64, libjpeg62-turbo:riscv64,

libjson-c5:riscv64, libk5crypto3:riscv64, libkeyutils1:riscv64, libklibc:riscv64, libkmod2:riscv64, libkrb5-3:riscv64, libkrb5support0:riscv64, liblcms2-2:riscv64, liblerc4:riscv64, libllvm17:riscv64, liblocale-gettext-perl, liblz4-1:riscv64, liblzma5:riscv64, libmd0:riscv64, libmnl0:riscv64, libmount1:riscv64, libncursesw6:riscv64, libnettle8:riscv64, lib-newt0.52:riscv64, libnftables1:riscv64, libnftnl11:riscv64, libnsl2:riscv64, libnspr4:riscv64, libnss3:riscv64, libnumber-compare-perl, libp11-kit0:riscv64, libpam-modules:riscv64, libpam-modules-bin, libpam-runtime, libpam-systemd:riscv64, libpam0g:riscv64, libpango-1.0-0:riscv64, libpangocairo-1.0-0:riscv64, libpangoft2-1.0-0:riscv64, libpcre2-8-0:riscv64, libpcsclite1:riscv64, libperl5.38:riscv64, libpixman-1-0:riscv64, libpng16-16:riscv64, libpopt0:riscv64, libproc2-0:riscv64, libpsl5:riscv64, libpthread-stubs0-dev:riscv64, libpy-thon3-stdlib:riscv64, libpython3.11-minimal:riscv64, libpython3.11-stdlib:riscv64, libread-line8:riscv64, librsvg2-2:riscv64, librsvg2-common:riscv64, libseccomp2:riscv64, libseli-nux1:riscv64, libsemanage-common, libsemanage2:riscv64, libsensors-config, libsen-sors5:riscv64, libsepol2:riscv64, libsharpyuv0:riscv64, libslang2:riscv64, libsm-dev:riscv64, libsm6:riscv64, libsmartcols1:riscv64, libsqlite3-0:riscv64, libss2:riscv64, libssl3:riscv64, libstdc++6:riscv64, libsystemd-shared:riscv64, libsystemd0:riscv64, libtasn1-6:riscv64, libtext-charwidth-perl:riscv64, libtext-glob-perl, libtext-iconv-perl:riscv64, libtext-wrapi18n-perl, libthai-data, libthai0:riscv64, libtiff6:riscv64, libtinfo6:riscv64, libtirpc-common, libtirpc3:riscv64, libudev1:riscv64, libunistring5:riscv64, libuuid1:riscv64, libvulkan1:riscv64, libwayland-client0:riscv64, libwebp7:riscv64, libwrap0:riscv64, libx11-6:riscv64, libx11-data, libx11-dev:riscv64, libx11-xcb1:riscv64, libxau-dev:riscv64, libxau6:riscv64, libxaw7:riscv64, libxcb-dri2-0:riscv64, libxcb-dri3-0:riscv64, libxcb-glx0:riscv64, libxcb-present0:riscv64, libxcb-randr0:riscv64, libxcb-render0:riscv64, libxcb-shape0:riscv64, libxcb-shm0:riscv64, libxcb-sync1:riscv64, libxcb-xfixes0:riscv64, libxcb1:riscv64, libxcb1-dev:riscv64, libxcompo-site1:riscv64, libxcursor1:riscv64, libxdamage1:riscv64, libxdmcp-dev:riscv64, libxdmcp6:riscv64, libxext6:riscv64, libxfixes3:riscv64, libxft2:riscv64, libxi6:riscv64, libx-inerama1:riscv64, libxkbfile1:riscv64, libxml2:riscv64, libxmu6:riscv64, libxmuu1:riscv64, libxpm4:riscv64, libxrandr2:riscv64, libxrender1:riscv64, libxshmfence1:riscv64, libxt-dev:riscv64, libxt6:riscv64, libxtables12:riscv64, libxtst6:riscv64, libxv1:riscv64, libxxf86dga1:riscv64, libxxf86vm1:riscv64, libxxhash0:riscv64, libz3-4:riscv64, libz-std1:riscv64, linux-base, linux-image-6.6.11-riscv64, linux-image-riscv64, login, logrotate, logsave, luit, mawk, media-types, mesa-vulkan-drivers:riscv64, mount, nano, ncurses-base, ncurses-bin, netbase, nftables, openjdk-22-jdk:riscv64, openjdk-22-jdk-headless:riscv64, openjdk-22-jre:riscv64, openjdk-22-jre-headless:riscv64, openssh-client, openssh-server, openssh-sftp-server, openssl, passwd, perl, perl-base, perl-modules-5.38, procps, publicsuf-fix, python3, python3-minimal, python3.11, python3.11-minimal, readline-common, rsync, runit-helper, sed, sensible-utils, shared-mime-info, strace, systemd, systemd-dev, systemd-sysv, sysvinit-utils, tar, tasksel, tasksel-data, tzdata, u-boot-menu, ucf, udev, unzip, usr-is-merged, usrmerge, util-linux, wget, whiptail, x11-common, x11-utils, x11proto-dev, xdg-user-dirs, xorg-sgml-doctools, xtrans-dev, zlib1g:riscv64

In addition, the smart home software stack includes a demonstration of several Apache StreamPipes extensions for IoT analytics developed for the smart home stack, showcasing the technical foundations of the edge client developed within WP4. This software consists of a lightweight microservice (written in either Java or Go), which can be deployed on resource-constrained hardware. The service includes adapters for various smart home protocols (e.g., BACNet, KNX, Modbus TCP/RTU/ASCII). Adapters can be instantiated from a centralized system using the Apache StreamPipes Connect library. This system allows to select one of the registered devices and directly deploys an adapter configuration to the edge node. Data gathered by the edge node is forwarded to the central core (which provides data management, persistence and analytics operators) over MQTT or NATS. This ensures continuous data streaming from resource-constrained edge devices to a central analytics interface. The complete software stack thus includes one or more extension services which are deployed directly

on edge nodes, the core as the central management and orchestration component, a time-series database and a message broker for persistence and live streaming, edge-centric. The system can also be used to integrate machine learning models using one of the integrated client libraries and the Python-based model zoo to integrate online machine learning models (e.g., using the RIVER framework), or reusing interchangeable model formats such as ONNX.

The main extensions developed within ISOLDE are the lightweight edge service (using AOT compilation for fast startup), the adapter implementation to connect with smart home protocols and the management extension to remotely instantiate and manage adapters from a central location at registered edge nodes.

5 Cellular IoT Demonstrator

5.1 Overview of the Demonstrator

The cellular IoT demonstrator (cIoT) aims to combine 5G cellular IoT standards for wireless communication with embedded HPC to build a modem that can connect to the internet through the cellular network while offering enough computing capabilities for embedded applications. The target markets of the proposed system are wearables, industrial monitoring, environmental sensing and monitoring, smart cities, and connectivity in automotive.

The embedded HPC part will be tackled with a 64-bit RISC-V system based on CVA6 that has been enhanced by “Ara": a powerful vector engine with a reconfigurable number of lanes. The vector capabilities of this embedded HPC subsystem will be particularly suited for compute-intensive workloads from signal processing or machine learning domains allowing IoT systems to be built that can complete demanding cognitive tasks enabled through recent AI solutions as well as complex control workloads at the edge, reducing the demand for energy costly communication and reducing the latency for control tasks. The goal is to build a programmable general-purpose host subsystem with a cellular network specialized domain which will be essential for building smart devices with reduced energy consumption (10 years of battery life) at low cost (single SoC integration).

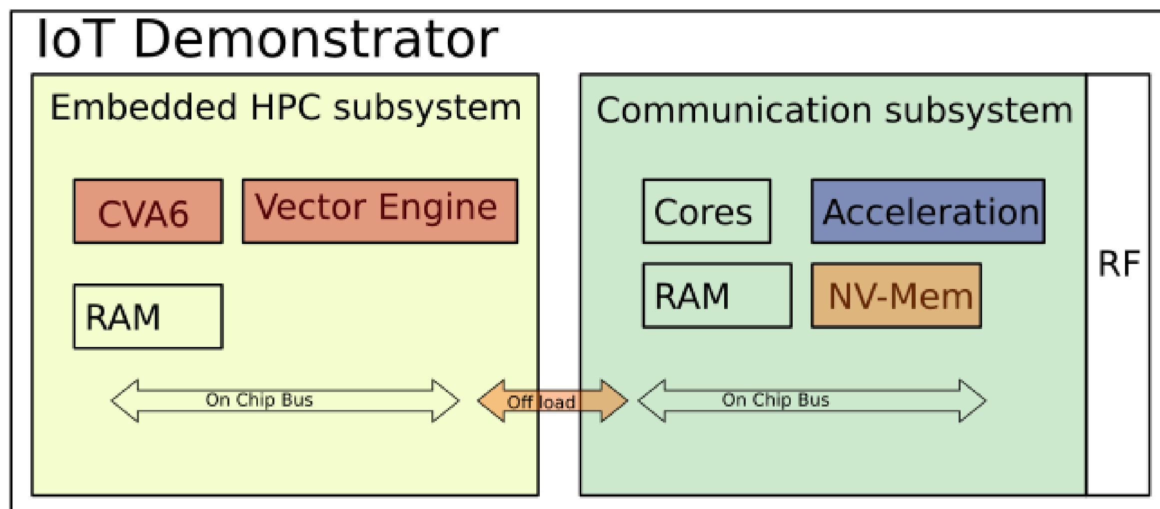


Figure 25: cIoT demonstrator overview: embedded HPC subsystem for capable of complex data processing tasks, coupled with a communication subsystem for cellular IoT connectivity

5.2 High-level Architecture

The general-purpose subsystem will be based on the Linux-capable CVA6. Alongside with CVA6, the SoC features multiple peripherals. A core local interrupt controller (CLIC) and platform local interrupt controller (PLIC) for timers and interrupts. A UART controller to support a serial console, a SPI controller to read a Linux image on a NOR flash, and a JTAG debug port to help bring up the system. All these peripheral IPs are chosen from open-source implementations. At this stage, the demonstration system will be realized on a Xilinx AMD based FPGA platform. For this implementation, a proprietary Xilinx AMD DDR4 controller will be used to access up to 4GB of shared RAM for the whole system. All the external chips (NOR & DRAM) will be taken from the chosen Xilinx AMD development board.

The embedded HPC and communication subsystem will work alone to allow for multiple clock domains and dedicated bus bandwidth on each side. Both parts of the system will be coupled to each other through a fully-digital double data-rate serial link. This scalable communication

interface is necessary to allow the demonstrator to be scaled into two different FPGA chips in case the system's area requires it.

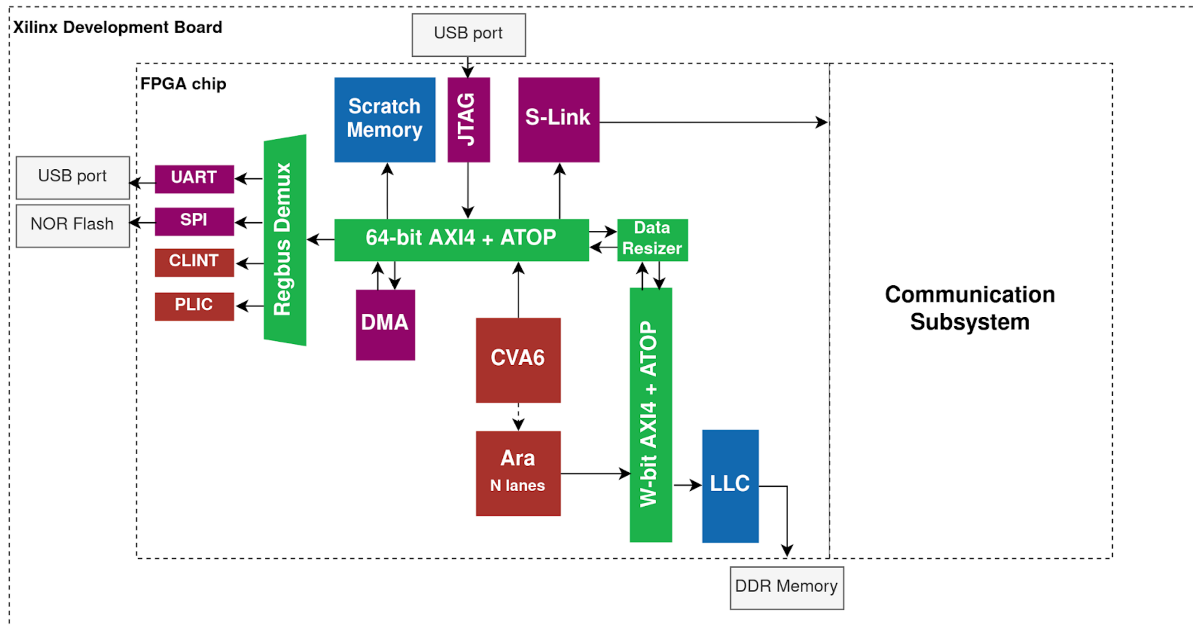


Figure 26: High-level overview of the IoT demonstrator

5.2.1 Embedded HPC Subsystem

The embedded HPC subsystem features Ara as a vector coprocessor. In order to provide superior performance when compared to a regular scalar core, Ara operates simultaneously on multiple 64-bit lanes in a Single Instruction Multiple Data (SIMD) fashion. Each lane contains its own Floating-Point Unit (FPU) and Vector Register File (VRF) chunk, which buffers the vector elements loaded from the DRAM via a Vector Load/Store Unit (VLSU). To avoid computational resource under-utilization, the AXI4 memory port of the VLSU scales linearly with the number of lanes (W = 4*Lanes [Byte]).

One of the challenges of integrating Ara and CVA6 in the same architecture is synchronizing and ordering the scalar and vector memory operations performed by the two independent scalar and vector LSUs. For example, to ensure cache coherence, a data-cache invalidation filter scans Ara's memory interface and, in the case of vector stores, invalidates the corresponding CVA6 data-cache lines.

To enable OS support in Ara, CVA6 MMU will be shared with the vector coprocessor to allow for address translation for vector memory operations. If the demonstrator runs in bare-metal mode, this feature can be disabled.

Since CVA6 and Ara's load/store units operate on different data widths, the embedded HPC subsystem will require two buses to operate. The 64-bit bus connects CVA6 with the peripherals, a small scratchpad memory used to load microcode, the serial link bridging to the communication domains and a Direct Memory Access (DMA) engine that can be used to fetch or push data to it. The W-bit bus contains only the DRAM accessible through a Last Level Cache (LLC). Any manager on the 64-bit bus can also access the DRAM through a data-width resizer. Both buses implement AXI4, but also the ATOP extensions defined in AXI5. All the elements cited above are shown above in Figure 26.

5.2.2 Communication Subsystem

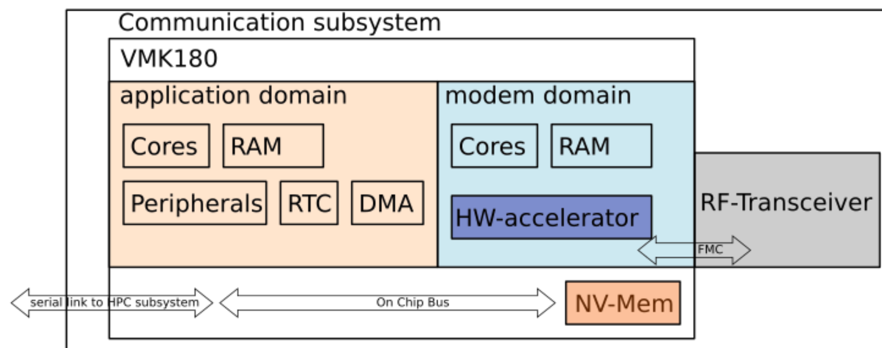


Figure 27: Communication Subsystem: Application domain, modem domain, RF-transceiver

The communication subsystem implements everything that is necessary to establish a cellular connection over LTE Cat1Bis or LTE NR. The subsystem consists of the following main blocks:

- Application domain: Hosts a set of peripherals (SPI, I2C, UART), local RAM, DMAs, and several processor cores that run higher level protocol software for Cat1Bis, and LTE NR and manage communication with the embedded HPC cluster through a serial link
- Modem domain: Hosts local RAM, several accelerators for signal processing tasks, and processor cores that run lower-level protocol software and control the different accelerators as well as the RF-transceiver
- RF transceiver: hosts the DFE, data interface and RF controller that allow to send and receive IQ samples

In addition, there is non-volatile storage for the binaries, and several interconnects.

5.2.2.1 FPGA prototype of the communication subsystem

The communication subsystem has been mapped to an FPGA for prototyping, verification and software development. A big board, the Xilinx Versal VMK180, has been chosen as a development board for the communication subsystem, because of its programmable logic size that allows it to map the full hardware on a single FPGA. On the prototyping platform, the non-volatile storage is emulated with DDR memory and the application and modem domain are mapped on to the programmable logic of the FPGA.

The RF-transceiver, as well as two antennas are connected to the FPGA over an adapter board that is connected over an FMC connector. In addition, there is a small extension board for USB UART.

Debug and trace functionality is supported through an internal JTAG master, and trace receiver IP that are implemented on the programmable logic and can be controlled over the processing system that runs Linux. Those two IPs can later be replaced with external debugger and tracer hardware once the chip has been fabricated.

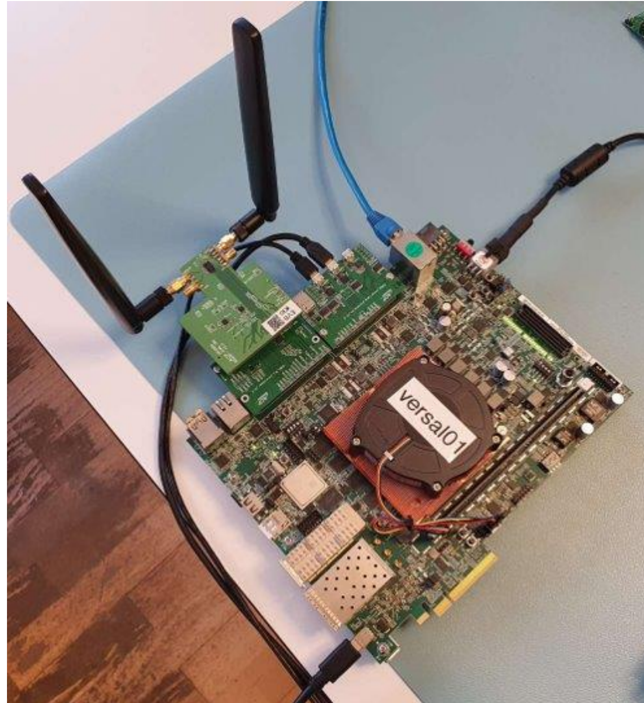


Figure 28: Image of the Xilinx Versal VMK180 board with the RF transceiver, two antennas, and an extension board for usb UART

5.2.3 Software

The presence of a Linux-capable core makes porting existing software and libraries to this SoC easy. In our demonstrator, CVA6 will run the version 6.1.22 Linux, along with the Busybox tools and executables. This includes for instance a Secure Shell (SSH) server. From Linux, our demonstrator will access the communication subsystem address space via a dedicated kernel module.

The communication subsystem will run a real time operating system on its processing system. The cluster will use the little kernel OS (https://github.com/littlekernel/lk). The protocol software as well as the lower-level software for the hardware accelerator control is under development.

Finally, devices within the Internet of Things, such as cars or mobile phones, can rely on the performances of Ara to accelerate compute intensive tasks like Deep Learning inference and signal processing. Without on-chip processing, the system would rely on cloud offloading which requires additional wireless communication thus power consumption. To demonstrate on-chip processing capabilities, we will benchmark floating point intensive operations on Ara. The operation will be selected from the BLAS (Basic Linear Algebra Subprograms) problems. BLAS [7] Is a specification implemented by several libraries, such as OpenBLAS [8]. It allows hardware vendors to provide a platform optimized implementation of each subprogram. Later, these can be called from higher level libraries like PyTorch [9] using a common API (Application Programming Interface).

5.3 Validation and Testing

The individual components of the embedded HPC and Communication subsystems are verified with stand-alone testbenches with standard rtl simulation tools. The subsystems are then verified with integration tests through a top level testbench to make sure that all components are properly connected to each other.

Performance in terms of speed will be primarily measured on the prototyping platform through performance counters that count the number of cycles, instructions, and stalls. This is especially important for the communication subsystem which has hard-real time constraints that

need to be fulfilled under any circumstances. To achieve the maximum throughput of the cellular protocol, which is 10mbps, the interface between the two domains must achieve the same throughput in both directions.

6 Conclusions

In this Deliverable we described the work done during the first year in the context of WP5.

The initial work focused on identifying and selecting the relevant IPs to be considered for each individual demonstrator, start discussing on possible integrability problems, propose one (or more) initial version of the demonstrator architecture, and define how and why the individual components should be used.

The Deliverable summarized the progress done for each Demonstrator (Section 2 for Space, Section 3 for Automotive, Section 4 for Smart Home, Section 5 for Cellular IoT). While the initial architectures proposed here may be revised at a later stage, this document already provides a clear overview and directions for future developments and can be taken as reference for the IPs development done in the context of WP2, WP3 and WP4.

7 References

[1] Ciancarelli, C. et al. (2023). Innovative ML-based Methods for Automated On-board Spacecraft Anomaly Detection. In: Ieracitano, C., Mammone, N., Di Clemente, M., Mahmud, M., Furfaro, R., Morabito, F.C. (eds) *The Use of Artificial Intelligence for Space Applications*. AII 2022. Studies in Computational Intelligence, vol 1088. Springer, Cham. https://doi.org/10.1007/978-3-031-25755-1_14

[2] Cheikh, A., Sordillo, S., Mastrandrea, A., Menichelli, F., Scotti, G., Olivieri, M.: Klessydra-t: Designing vector coprocessors for multithreaded edge-computing cores. IEEE Micro 41(2), 64–71 (2021). https://doi.org/10.1109/MM.2021.3050962

[3] Cota, E.G., Mantovani, P., Di Guglielmo, G., Carloni, L.P.: An analysis of accelerator coupling in heterogeneous architectures. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (2015). https://doi.org/10.1145/2744769.2744794

[4] ONNX: ONNX 1.17.0 documentation. https://onnx.ai/onnx/operators/onnx__Conv.html (2024), accessed on April 4, 2024

[5] ARM: ARM1176JZ-S Technical Reference Manual r0p7. https://developer.arm.com/documentation/ ddi0333/h/coprocessor-interface/coprocessor-pipeline/ coprocessor-instructions (2024), accessed on April 4, 2024

[6] Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.: The molen polymorphic processor. IEEE Transactions on Computers 53(11), 1363–1375 (2004). https://doi.org/10.1109/TC.2004.104

[7] Blackford, L.S. et al., 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2), pp.135–151.

[8] http://www.openblas.net/

[9] Adam Paszke et al.: Automatic differentiation in PyTorch. NIPS (2017)

8 Annex

8.1 conv2d_api.cpp

```
typedef int v4xi32 __attribute__ (( vector_size (4 * sizeof ( int ))));
// accelerator 's register map
const int reg_base = 0 x8000000 ;
const int reg_y = reg_base + 4;
const int reg_y_shape = reg_y + 4;
const int reg_x = reg_y_shape + 4*4;
const int reg_x_shape = reg_x + 4;
const int reg_w = reg_x_shape + 4*4;
const int reg_w_shape = reg_w + 4;
const int reg_pad = reg_w_shape + 4*4;
const int reg_str_dil = reg_pad + 4*4;
inline v4xi32 __conti_onnx_conv2df32 ( float * y
                                      , float * x
                                      , v4xi32 x_shape
                                      , float * w
                                      , v4xi32 w_shape
                                      , v4xi32 pad
                                      , v4xi32 str_dil ){
        int * cfg = ( int *) reg_y ;
        * cfg = reinterpret_cast < int >( y );
        // input
        cfg = ( int *) reg_x ;
        * cfg = reinterpret_cast < int >( x );
        cfg = ( int *) reg_x_shape ;
        * cfg = reinterpret_cast < int >(& x_shape );
        // weight
        cfg = ( int *) reg_w ;
        * cfg = reinterpret_cast < int >( w );
        cfg = ( int *) reg_w_shape ;
        * cfg = reinterpret_cast < int >(& w_shape );
        // padding
        cfg = ( int *) reg_pad ;
        * cfg = reinterpret_cast < int >(& pad );
        // striding dilation
        cfg = ( int *) reg_str_dil ;
        * cfg = reinterpret_cast < int >(& str_dil );
        // start the operation
        cfg = ( int *) reg_base ;
        * cfg = 0 xabc ;
        v4xi32 y_shape ={1 ,2 ,3 ,4};
        return y_shape ; }
```

```
extern float x [1][1][5][5];
extern float w [1][1][3][3];
extern float y [1][1][3][3];
int main (){
v4xi32 x_shape ={1 ,1 ,5 ,5};
v4xi32 w_shape ={1 ,1 ,3 ,3};
v4xi32 pad ={0 ,0 ,0 ,0}; // padding
v4xi32 str_dil ={1 ,1 ,0 ,0}; // stride_dilation
v4xi32 y_shape ;
y_shape = __conti_onnx_conv2df32 (( float *) y
                                  ,( float *) x
                                  , x_shape
                                  ,( float *) w
                                  , w_shape
                                  , pad
                                  , str_dil );
                                  return 0;
                                  }
```

Generated instructions flow:

```
1     addi sp , sp , -64
2     auipc a0 , % got_pcrel_hi ( y)
3     lw a0 , % pcrel_lo (. Lpcrel_hi0 )( a0 )
4     lui a1 , 32768
5     auipc a2 , % got_pcrel_hi ( x)
6     lw a2 , % pcrel_lo (. Lpcrel_hi1 )( a2 )
7     sw a0 , 4( a1 )
8     auipc a0 , % got_pcrel_hi ( w)
9     lw a0 , % pcrel_lo (. Lpcrel_hi2 )( a0 )
10    sw a2 , 24( a1 )
11    addi a2 , sp , 48
12    sw a2 , 28( a1 )
13    sw a0 , 44( a1 )
14    addi a0 , sp , 32
15    sw a0 , 48( a1 )
16    addi a0 , sp , 16
17    sw a0 , 64( a1 )
18    mv a0 , sp
19    sw a0 , 80( a1 )
20    lui a0 , 1
21    addi a0 , a0 , -1348
22    sw a0 , 0( a1 )
23    li a0 , 0
24    addi sp , sp , 64
25    ret
```

8.2 conv2d.cpp

```
extern float x [1][1][5][5];
extern float w [1][1][3][3];
extern float y [1][1][3][3];
int main (){
    v4xi32 x_shape ={1 ,1 ,5 ,5};
    v4xi32 w_shape ={1 ,1 ,3 ,3};
    v4xi32 pad ={0 ,0 ,0 ,0}; // padding
    v4xi32 str_dil ={1 ,1 ,0 ,0}; // stride_dilation
    v4xi32 y_shape ;
    y_shape = __builtin_onnx_conv2df32 (( float *) y
                                        ,( float *) x
                                        , x_shape
                                        ,( float *) w
                                        , w_shape
                                        , pad
                                        , str_dil );
    return 0;
}
```

Invoking the compiler,

clang - cc1 -S

- triple riscv32 - unknown - elf - target - feature +v

- target - abi ilp32d - O3 -o -

conv2d . cpp

will generate the following flow of instructions:

```
1    vle32 .q Q0 , 1, 1, 0, 0
2    auipc a0 , % got_pcrel_hi ( w)
3    lw a0 , % pcrel_lo (. Lpcrel_hi0 )( a0 )
4    auipc a1 , % got_pcrel_hi ( x)
5    lw a1 , % pcrel_lo (. Lpcrel_hi1 )( a1 )
6    auipc a2 , % got_pcrel_hi ( y)
7    lw a2 , % pcrel_lo (. Lpcrel_hi2 )( a2 )
8    vle32 .q Q1 , 0, 0, 0, 0
9    vle32 .q Q2 , 1, 1, 3, 3
10   vle32 .q Q3 , 1, 1, 5, 5
11   conv2d . f32 a2 , Q0 , a1 , Q3 , a0 , Q2 , Q1 , Q0
12   li a0 , 0
13   ret
```