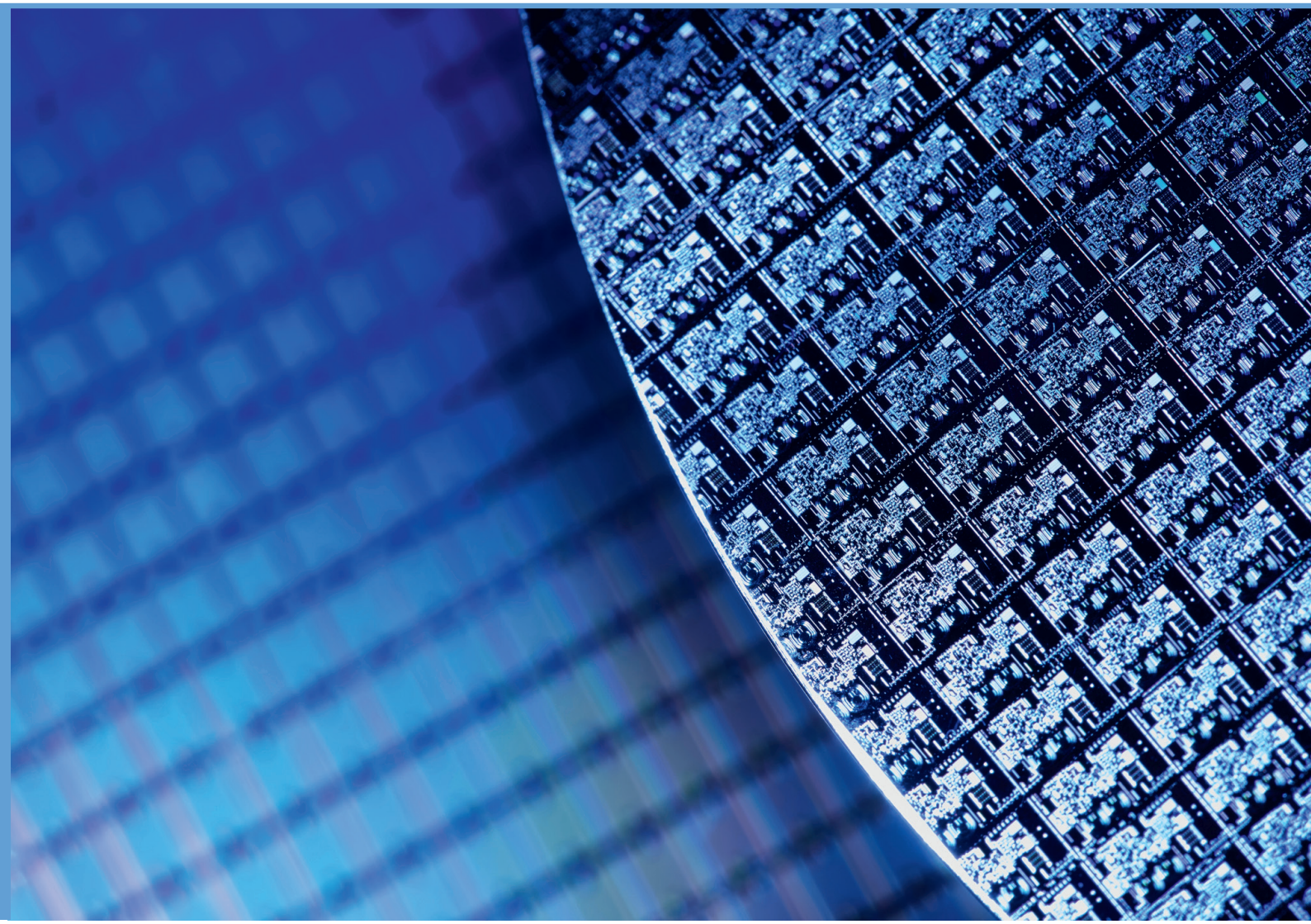
RIGA TECHNICAL UNIVERSITY

Rihards Novickis

IMPLEMENTATION OF STEREO-VISION ALGORITHMS IN HETEROGENEOUS EMBEDDED SYSTEMS

Doctoral Thesis



RTU Press
Riga 2022

RIGA TECHNICAL UNIVERSITY
Faculty of Electronics and Telecommunications
Institute of Radio Electronics

Rihards Novickis
Student of the Doctoral study program ”Electronics”

Implementation of stereo-vision algorithms in heterogeneous embedded systems

Doctoral Thesis

Academic supervisors
Dr.sc.Ing., senior researcher
Artūrs Ābolti̧š

Dr.sc.Ing., researcher
Rolands Šāvelis

Riga 2022

A DOCTORAL THESIS SUBMITTED TO RIGA TECHNICAL UNIVERSITY IN FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF SCIENCE IN ENGINEERING

The public defense of the doctoral thesis for promotion to the doctor's degree in engineering will take place on 11th of March, 2022, at 13:00 in Riga Technical University, 12 Azenes Street, Riga.

OFFICIAL REVIEWERS

Associate Professor Dr. sc. ing. Dmitrijs Pikuļins, Riga Technical University, Latvia

Dr.sc.ing., Assistant Professor Paolo Meloni, University of Cagliari

Dr.sc.ing., Associate Professor Wassim Hamidouche, University of Rennes

CONFIRMATION

I confirm that this doctoral thesis, submitted for a degree in engineering at the Riga Technical University, is my own work. The doctoral thesis has not been submitted for a degree in any other university.

Rihards Novickis . . . . . (Signature)

Date: . . . . . .

The doctoral thesis is written in English, contains introduction, 4 chapters, conclusions, references, 10 appendices, index, 76 Figures and 7 Tables, 122 pages in total. The list of references consists of 132 titles.

ABSTRACT

Despite the rapid development, hardware's performance increase saturates; therefore, new approaches in computing are necessary. This work is part of a global effort to solve the saturation challenge of Moore's Law by drawing attention to the increasing complexity of modern processing systems, applying Heterogeneous System on Chip (HSoC) technology and implementing stereo vision algorithms. The limiting factor for this application is the high complexity of the development process that is determined by the variety of involved development paradigms, i.e. the design of digital circuits, designing internal communications, developing software for hardware control, dealing with memory virtualization and deploying system architectures.

To achieve the set objective, the following themes are examined: heterogeneous embedded system development approaches, a combination of different abstraction levels, effective internal communications, co-processing using different computing hardware, real-time processing, implications for modern AI-based algorithms and realization of an efficient computer vision pipeline. The main contributions of this thesis can be summarized as follows:

- A collection of methods and approaches for the system-level design of HSoCs.
- Method for maximizing throughput of Feed Forward Neural Network processing pipeline.
- HSoC-based architecture for computer vision processing.

The thesis is useful for future system architects, computer vision researchers, SoC designers and engineers tackling the design of HSoCs. The findings and conclusions of the thesis are manifested in the form of a stereo-vision demonstrator.

The doctoral thesis is the result of the research carried out at the Institute of Electronics and Computer Science (EDI) within the following *Horizon2020 ECSEL* projects: "Integrated Components for Complexity Control in affordable electrified cars" (3Ccar, GA:662192), "Intelligent Motion Control Platform for Smart Mechatronic Systems" (I-MECH, GA:737453), "Programmable Systems for Intelligence in Automobiles" (PRYSTINE, GA:783190), "Advanced packaging for photonics, optics and electronics for low-cost manufacturing in Europe" (APPLAUSE, GA:826588), "Framework of Key Enabling Technologies for Safe and Autonomous Drones" (COMP4DRONES, GA:826610).

The work consists of 122 pages, 76 figures, 7 tables, 132 sources of literature and 10 appendixes.

ANOTĀCIJA

Neskatoties uz tehnoloģiu straujo attīstību ir novērojams piesātinājums aprēķinu aparatūras veiktspējas pieaugumā, līdz ar to ir nepieciešamas jaunas pieejas aprēķinu veikšanai. Šis darbs ir daļa no globāliem centieniem rast risinājumu Mūra likuma piesātinājuma izaicinājumam, vēršot uzmanību pieaugošajai sarežģītībai mūsdienu aprēķinu sistēmās, pielietojot uz Programmējamiem Loģikas Masīviem balstītas heterogēnas vienkristāla sistēmas un īstenojot stereo redzes algoritmus. Viens no šāda pilietojuma galvenajiem limitējošiem faktoriem ir izstrādes procesa augstā sarežģītība, kuru nosaka daudzās iekļautās izstrādes paradigmas, t.i. digitālo shēmu projektēšana, iekščipa komunikāciju nodrošināšana, aparatūras kontroles programmatūras izstrāde, atmiņas virtualizācijas mehānismu izmantošana un sistēmas arhitektūras realizācija.

Lai sasniegtu izvirzīto mērķi, tiek izskatīti sekojošie temati: heterogēno iegulto sistēmu izstrādes pieejas, atšķirīgu abstrakcijas līmeņu apvienošana, efektīvas iekščipa komunikācijas, dažādas aprēķinu aparatūras līdzapstrāde, reāla laika apstrāde, implikācijas mūsdienu, uz mākslīgā intelekta balstītu, algoritmu realizācijai un efektīvas datorredzes apstrādes ķēdes.

Galvenie šīs disretācijas pienesumi ir:

- metožu un pieeju krājums heterogēnu iegultu sistēmu izstrādei;
- metode dziļo neironu tīklu aprēķinu virknes caurlaidspējas palielināšana;
- heterogēnu iegultu sistēmu arhitektūra datorredzes algoritmu realizācijai.

Disertācija ir noderīga sistēmu arhitektiem, datorredzes pētniekiem, vienkristāla sistēmu izstrādātājiem, kā arī inženieriem, kuri nodarbojas ar heterogēnām iegultām sistēmām. Šī darba galvenie atradumi un secinājumi pielietoti reāla laika stereo redzes gala demonstrātora izveidē.

Disertācija ir izstrādāta Elektronikas un datorzinātņu institūtā (EDI) sekojošu iniciatīvas Horizon 2020 ECSEL kopuzņēmuma projektu ietvaros: ”Integrētas komponentes sarežģītības kontrolei pieejamos elektrificētos transporta līdzekļos” (3Ccar, GA:662192), ”Inteliģenta kustību kontroles platforma viedām mehatroniskām sistēmām” (I-MECH, GA:737453), ”Programmējamas Sistēmas Inteliģencei Automobiļos” (PRYSTINE, GA:783190), ”Uzlabota fotonikas, optisko un elektroniko komponenšu iepakošana/montāža zemu izmaksu ražošanai Eiropā” (APPLAUSE, GA:826588), ”Pamattehnoloģiju ietvars drošu un autonomu dronu lietojumiem” (COMP4DRONES, GA:826610).

Darbā ir 122 lapaspuses, 76 attēli, 7 tabulas, 132 izmantotie literatūras avoti un 10 pielikumi.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisors Dr Artūrs Āboltiņš and Dr Rolands Šāvelis for their patient guidance, encouragement and useful critiques.

I would also like to express my gratitude to my scientific director Dr Modris Grietāns and my friend and colleague Dr Kaspars Ozols for providing this research opportunity and broadening my horizons with international project opportunities. You have always provided support and advice during my creative mess. That surely would not be possible without you.

A special thanks to my mentor and friend, Mihails Pudžs. I still have not met anyone with such appreciation, fascination and never-ending lust for the research process. Thanks to you, I have found that passion and discipline can coexist. That surely would not have been such an amazing experience. Special thanks also go to my friend and technical guru Rinalds Ruskulis. Our discussions many times have facilitated the broadening of my horizons.

I thank all my colleagues at the *Institute of Electronics and Computer Science* (EDI) for the friendly and motivating environment. You have been like a family to me. Specifically, I would like to express my deepest gratitude to the EDI System on Chip research group. Our whiteboard discussions have always made my work a pleasure, as well as our drive towards our common ambitious goals. Special thanks go to the head of my laboratory - Dr Roberts Kadiķis. I admire your ability to make far-reaching decisions and, of course, your deep understanding of topics of philosophy, politics and history.

Also, thanks to the staff of Riga Technical University for providing this great opportunity of discovering the fascinating world of electronics and special thanks to Dr Dmitrijs Pikuļins for keeping me motivated throughout the process.

Last but not least, I would like to thank my family. My brothers - Oskars Novickis for commending me to study electronics and Maris Novickis for introducing me to the fascinating world of computers. And of course, I am very grateful to my parents Veneranda Novicka and Vilhelms Novickis. I praise you for your encouragement, understanding and never-ending support. You have always been there, no matter how bad things may have become.

Rihards Novickis,

Riga 2022

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

3Ccar Integrated Components for Complexity Control in affordable electrified cars. 21, 68

ACP Accelerator Coherency Port. 54, 57

ADC Analog-to-Digital Converter. 59

AI Artificial Intelligence. 18, 19, 74, 99, 100

ALU Arithmetic Logic Unit. 23

AMBA Advanced Microcontroller Bus Architecture. 30, 32, 55

AMD Advanced Micro Devices. 85

AMP Asynchronous Multi-Processing. 19, 51, 58–64, 99

ANN Artificial Neural Network. 46, 70, 71, 100

APPLAUSE Advanced packaging for photonics, optics and electronics for low cost manufacturing in Europe. 21, 75

ARM Acorn RISC Machine. 30, 32, 53–55, 60

ASIC Application Specific Integrated Circuit. 25, 26, 28, 97

ASIP Application-Specific Instruction-set Processors. 27–29

AXI Advanced eXtensible Interface. 30, 73

BPF Berkeley Packet Filter. 34

CCD Charge-Coupled Device. 39

CFA Color Filter Array. 80

CGRA Coarse-Grained Reconfigurable Array. 27, 28

CLA-FPU Control Law Accelerator Floating Point Unit. 59

CMA Contiguous Memory Allocator. 53, 56

CMOS Complementary Metal-Oxide Semiconductor. 82

CNN Convolutional Neuron Networks. 48

COMP4DRONES Framework of Key Enabling Technologies for Safe and Autonomous Drones. 64, 67, 75

CPU Central Processing Unit. 9, 24, 27, 30, 31, 35, 36, 52, 59, 60

CUDA Compute Unified Device Architecture. 28

DDR Double Data Rate. 57, 61, 79

DL Deep Learning. 46

DMA Direct Memory Access. 31, 36, 53–56, 73, 76, 78, 99

DNN Deep Neural Network. 25, 46

DSP Digital Signal Processing. 24, 31

ECSEL Electronic Components and Systems for European Leadership. 21, 58, 64

EDA Electronics Design Automation. 17

ELF Executable Linked Format. 60, 63, 65

FFNN Feed Forward Neural Network. 19, 20, 48, 68–71, 73, 74, 100

FIFO First-In-First-Out. 62, 63, 67, 79, 80

FPGA Field Programmable Gate Array. 19, 24–26, 28, 31, 32, 36, 51–58, 60, 69, 74, 76, 82, 97–100

FPSoC Field Programmable System on Chip. 53

GPU Graphics Processing Unit. 9, 27, 28, 31, 32, 52, 74

H2020 Horizon 2020. 21, 58, 64, 65

HDL Hardware Description Language. 25, 26

HLS High-Level Synthesis. 20, 27, 53, 68, 71, 73, 74

HPC High Performance Computing. 24, 25

HPS Hard Processing System. 54

HSA Heterogeneous System Architecture. 51

HSF Heterogeneous Systems Foundation. 51

HSoC Heterogeneous System on Chip. 9, 18–20, 22, 29, 31, 32, 34, 36, 51–53, 59, 61, 68, 75, 99

I-MECH Intelligent Motion Control Platform for Smart Mechatronic Systems. 21, 58, 59

I/O Input/Output. 24

IC Integrated Circuit. 17, 25, 29, 30

IOMMU Input/Output Memory Management Unit. 36

IoT Internet of Things. 98

IP Intellectual Property. 10, 73, 86

IPC Inter-Process Communication. 35, 67

IPI Inter-Processor-Interrupt. 62

ISA Instruction Set Architecture. 22, 23, 29, 31, 32

LUT Look-Up-Table. 72

MIMO Multiple-Input, Multiple-Output. 58

ML Machine Learning. 37, 46, 68

MM Memory-Mapped. 73

MMU Memory Management Unit. 33, 35, 60, 63

MPU Micro Processing Unit. 73, 76, 99

MSGDMA Modular Scatter-Gather Direct Memory Access. 54–57

NN Neural Network. 19, 46, 68, 69, 71

NoC Network on Chip. 30

OCRAM On-Chip Random Access Memory. 53, 61, 68, 82, 88

OCROM On-Chip Read Only Memory. 61

OpenCL Open Computing Language. 28

OpenGL Open Graphics Library. 76

OS Operating System. 18, 19, 34, 36, 52

PC Program Counter. 23, 97

PCIe Peripheral Component Interconnect Express. 20, 31, 36, 52, 75, 97

PPI Programmable Peripheral Interconnect. 51

PRYSTINE Programmable Systems for Intelligence in Automobiles. 21, 64, 67

PWM Pulse Width Modulation. 59

RAM Random-Access Memory. 54

ROS Robot Operating System. 65, 67

RT Real-Time. 19, 60–62

RTL Register Transfer Level. 18, 25–27, 95, 96

SCU Snoop Control Unit. 56, 61, 73

SDRAM Synchronous Dynamic Random-Access Memory. 54

SIMD Single Input Multiple Output. 53

SIP Silicon Intellectual Property. 29, 30, 32, 77, 100

SIPO Serial-In-Parallel-Out. 95

SoA State of Art. 22, 46, 59, 65, 68, 74, 82, 99

SoC System on Chip. 17, 18, 22, 28–32, 51, 54, 57, 60, 78, 97, 99, 100

SRAM Static Random Access Memory. 82

ST Streaming. 73, 84

TLB Translation Lookaside Buffer. 35

TSMC Taiwan Semiconductor Manufacturing Company. 29

UAV Unmanned Aerial Vehicle. 19

USB Universal Serial Bus. 31

VHDL Very High Speed Integrated Circuit Hardware Description Language. 97

VLSI Very Large Scale Integration. 17

WTA Winner-Take-All. 45

ZMQ Zero Message Queue. 67

INTRODUCTION

During his or her lifetime an experienced digital *Integrated Circuit* (IC) engineer has experienced the incredible journey of digital design automation and the rise of *Very Large Scale Integration* (VLSI), hence a migration from a very manual work-driven design of the digital chip to the highly sophisticated *Electronics Design Automation* (EDA) tools, which can deal with the placement and routing of billions of transistors. Moore's law [1] has become something more than just a prediction, it guides the intimate relationship between the innovation and modern semiconductor companies. The public expects Improved performance and innovative features while investors anticipate the new technologies.

The demand for higher performance yet lower-cost processing and standardization of on-chip digital communication protocols have led to the emergence of revolutionary computing system paradigm - *System on Chip* (SoC) [2]. The SoC is an IC which incorporates almost all components of an electronic system into a single chip. This brings notable improvements to power efficiency, inter-component communication bandwidth and computing power despite increasing the complexity of the overall system. Most notably, SoC technology has led to the rise of smartphones and is being adopted widely across the industries.

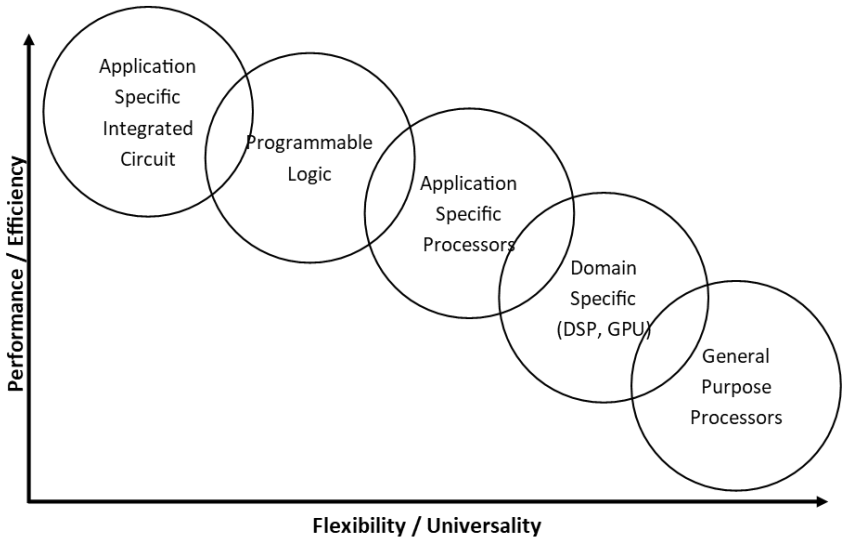


Figure 0.1. Comparison between implementation platforms.

Naturally, the computational characteristics and requirements of different applications have led to the specialization of hardware and involves the trade-off between computational platform's universality and performance, thus leading to a range of technologies: general-purpose

processors, domain-specific processors, application-specific processors, programmable logic and specialized integrated circuits. Even more difficult-to-meet performance and efficiency requirements drove the establishment of an even more complicated system category - *Heterogeneous System on Chip* (HSoC) [3]. Now a single chip incorporates a variety of sequential and parallel computational paradigms promising even higher performance, expectedly, on the expense of complexity.

Meanwhile, the advancement of modern *Artificial Intelligence* (AI) based algorithms transforms many fields and is the key-technology behind many innovative applications [4]. With this new prospect of computerized systems being able to perceive the world reliably, we must ask an important question: *How do we facilitate the widespread of these smart systems?* Apart from the algorithms, the perception challenge concerns power consumption, communication bandwidth, real-time performance and time-to-market considerations of the product.

The contemporary SoC technologies bare potential of solving the perception challenge in a commercially feasible way, but this still requires solving challenges of system design involving multiple computational paradigms and abstractions ranging from *Register Transfer Level* (RTL) design to the level of *Operating System* (OS) and even extending to an organization of systems. Such aspects as algorithm partitioning across different computational paradigms, reliable on-chip communications' architecture and compliance with real-time control system performance are still a major challenge [5, 6].

Considering the above mentioned, **this thesis is addressing the relationship between computerized perception and increasingly complex HSoC technologies**. Particularly, the on-chip hardware and software co-architectures, implementation of stereo-vision and AI algorithms and associated real-time considerations. The primary aim of this thesis is to develop and improve computer vision development techniques and methods for HSoC technology.

Several tasks have been defined in order to reach the aim of the thesis:

1. identify methods for complementing RTL and software-based computing paradigms;
2. design heterogeneous architectures and tools for utilizing heterogeneous SoC technology;
3. design heterogeneous approach to the implementation of image processing pipelines;
4. implement and conduct experimental research on the developed tools and algorithms;
5. draw conclusions about the results of this Thesis.

The main contributions of this thesis can be summarized as:

1. **A collection of methods and approaches for system-level design of HSoC technologies.** The design of HSoC-based technologies is a relatively complex task as it involves multiple levels of abstraction. The complexity must be addressed in order to enable an efficient processing of stereo-vision algorithms. These methods serve as a foundation for the rest of the thesis as they involve the primary on-chip interfacing mechanisms and requirements for software and hardware co-development flow.

   The thesis suggests an *Field Programmable Gate Array* (FPGA) master-based system architecture suitable for implementing computer vision algorithms. A shortcoming in the characterization of different HSoCs has been identified and corrected accordingly by performing bench-marking of on-chip communication mechanisms. The developed system architecture adopts Linux OS as it provides a good code base that enhances the overall capabilities of the system, although with the caveat of increased complexity.

   Furthermore, the *Real-Time* (RT) performance of HSoCs is addressed by developing an *Asynchronous Multi-Processing* (AMP) subsystem approach. In the proposed approach, at least one of the processor cores is dedicated to a RT application, while the rest of the system runs Linux. This method aspires to keep both the RT performance of a bare-metal application and the available software code base of Linux while sacrificing simplicity.

   Another identified shortcoming is a lack of software component management frameworks suitable for embedded systems. This need is addressed by developing modular software component management (*compage*) and communication (*icom*) frameworks. These frameworks simplify prototyping activities for systems where RT performance is essential. At the moment of writing this thesis, the developed frameworks are already utilized for autonomous driving and *Unmanned Aerial Vehicle* (UAV) applications.

2. **Method for maximizing the throughput of *Feed Forward Neural Network* (FFNN) processing pipeline.** In recent years, many algorithm benchmarks, including stereo-vision, are being dominated by the approaches based on AI. Therefore it is crucial to understand their positioning in the context of modern HSoC technologies.

   The developed novel approach enables the design of a throughput-optimized processing pipeline for FFNNs. It reexamines the *Neural Network* (NN) implementation challenge and restates the description of FFNNs to adapt it for pipelining. The network is split

into elementary layers, where each layer is associated with an abstract resource. These resources can be allocated to each of the layers and determine the delay characteristics of the whole pipeline.

The approach accommodates a tool, which converts topology into high-level code for *High-Level Synthesis* (HLS) pipeline. Tool's inputs are the topology of the network and target latency for a single stage of the pipeline (elementary layer). The developed method is suitable for virtual sensor implementation, especially when a high sample rate is required.

3. **HSoC-based architecture for computer vision processing.** To fully comprehend the significance of employing HSoC technologies for computer vision algorithm implementation, a stereo correspondence matching algorithm has been implemented using engineering design methods.

   The developed correspondence matching pipeline is fully implemented in the programmable logic. It consists of deinterleaving, Bayers pattern interpolation, lens distortion correction, rectification, feature extraction and correspondence matching. The processing system is used mainly for accelerator control, image acquisition via *Peripheral Component Interconnect Express* (PCIe) bus and transfer of the resulting image to the demonstrator system.

   During the writing of this thesis, commercially viable stereo cameras have entered the market. Nevertheless, there are still opportunities for improvement. One of the most formidable achievements is the design of a fully pipelined image transformation circuit, which corrects lens distortions and performs perspective transformation of images. The design has been accommodated with a novel approach for parallel access of N-dimensional data in digital logic while conserving memory utilization.

This work examines the following theses:

1. **The developed Asynchronous Multi-Processing subsystem solution enables exploiting functionality of a modern operating system while supporting real-time processing with control-loop latencies' jitter not exceeding a standard deviation of 1 ms.**

2. **The developed throughput-optimized FFNN design method offers better performance when compared to neiron-centric approaches (2-30 times) and methods following**

RTL design flow (>1000 times).

3. **The HSoC technology is suitable for implementing image processing pipelines in a fully pipelined manner achieving performance appropriate for modern real-time systems, whith a control loop's length less than 50 ms for <1.5 MP images.**

The thesis organization is as follows. Section 1 summarises digital computing paradigms and principles, and Section 2 depicts image processing principles and different stereo-vision methods and techniques. Section 3 describes the design of HSoC-based system architectures and their real-time characteristics. Section 4 deals with the challenge of adapting and implementing computer vision algorithms in HSoC-based technologies. Section 5 presents the conclusions of the thesis.

The results presented in this thesis are mostly the result of research activities in *Horizon 2020* (H2020) *Electronic Components and Systems for European Leadership* (ECSEL) *Integrated Components for Complexity Control in affordable electrified cars* (3Ccar)[1], *Intelligent Motion Control Platform for Smart Mechatronic Systems* (I-MECH)[2], *Programmable Systems for Intelligence in Automobiles* (PRYSTINE)[3], *Advanced packaging for photonics, optics and electronics for low cost manufacturing in Europe* (APPLAUSE)[4] projects and are also presented in several published [7–13], accepted [14] and forming [15–17] papers.

[1] https://www.3ccar.eu/
[2] https://www.i-mech.eu/
[3] https://prystine.eu/
[4] https://applause-ecsel.eu/

1. TECHNOLOGICAL CONTEXT

This section briefly lays out the technological principles necessary to comprehend contributions to the *State of Art* (SoA) further depicted in sections 3 and 4. An experienced professional can skip this section or parts of it if the respective topics are familiar. The discussed principles cover the basics of processors and programmable logic, the principles and significance of SoC and HSoC technologies and such essential concepts as the Linux operating system and memory virtualization - a core principle in multiple master system design.

1.1 Sequential Processing

The definition of Entscheidungsproblem[5] by David Hilbert and Wilhelm Ackermann [18], and the attempts to solve it led to the formalization of algorithms and the establishment of early computational models, most notably the emergence of Alan Turing's computing machine [19] in 1936. Turing's automatic machine model brought the prospect of automated and programmable systems capable of executing algorithms, and it took less than a decade for John von Neumann to develop a computer architecture [20] which in many ways still characterizes modern computers. Since the first microprocessor[6], computers are becoming an integral part of technology and everyday life.

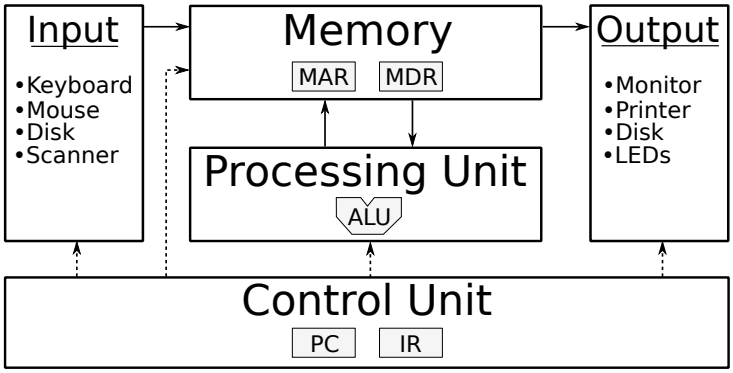


Figure 1.1. The von Neumann model [21] (*MAR* - Memory Address Register, *MDR* - Memory Data Register, *PC* - Program Counter, *IR* - Instruction Register, *ALU* - Arithmetic and Logic Unit).

A microprocessor's functionality is fully characterized by the instruction set that it is capable of executing, which is also called the *Instruction Set Architecture* (ISA). ISA has been defined

[5]German for "decision problem"
[6]Intel 4004, introduced in 1971

as a contract between software and hardware, which allows independent development of both. ISA also ensures software portability as a program written for the particular ISA can run on many processor generations. Unfortunately, this benefit also makes migration to a new ISA very difficult, as ISAs with a larger software base (not necessarily better design) tend to persist longer. Besides serving as a reference to the compiler developers, ISA also serves as the specification for the processor designers. Most often, it defines a set of instructions called *assembly instructions*, which are then provided by the microarchitectures. ISAs tend to evolve very slowly due to the inertia against recompiling and redeveloping software. Consequently, innovation tends to be on the side of microarchitectures [22].

The main computing characteristics of the processor can be comprehended by examining the generic instruction cycle, which typically consists of 5 phases [22, 23].

- *Instruction fetch (IF)*. The *Program Counter* (PC) register is used to retrieve the next instruction from the memory and load it into the instruction register. Simultaneously the PC register is incremented for the next *IF* cycle.
- *Instruction decode (ID)*. The instruction is examined in order to generate the necessary control signals for the instruction execution phase.
- *Operand(s) fetch (OF)*. Usually the instruction specifies one or more operands, which are needed to be fetched before the actual instruction execution.
- *Instruction execution (EX)*. Once the necessary operands are available, the decoded instruction generates internal control signals, e.g. to perform an arithmetic operation using *Arithmetic Logic Unit* (ALU). Depending on the complexity of the operation, this phase can execute variable number of clock cycles.
- *Operand store (OS)*. The result of the instruction can be stored in a register or memory location depending on the specified addressing mode.

The complexity and latency vary for different instructions and phases, therefore actual instruction pipelines are usually separated into more balanced sub-stages. Balanciation of the instruction pipeline's stages, unification of instruction types and minimization of pipeline stalls are three primary design challenges of a modern processor [22].

The instruction cycle encapsulates the intrinsic advantage of the sequential processing paradigm. Most notably, the system is universal and can be used to implement almost any kind of algorithm.

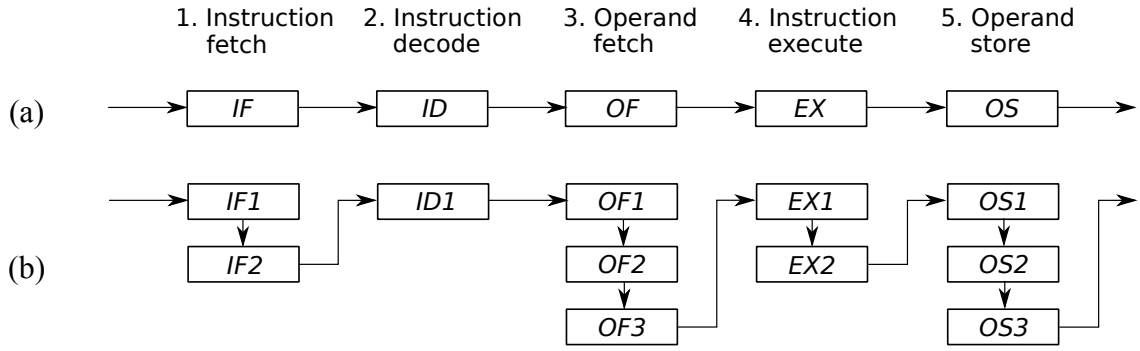


Figure 1.2. a) A generic five-stage instruction pipeline. (b) Instruction pipeline balanced into 11 stages, which correspond to clock cycles.

This feature has led to a wide application of processors ranging from simple keyboard controllers to massive *High Performance Computing* (HPC) servers. Nevertheless, the intrinsic weaknesses of this processing paradigm are the memory access irregularities and inter-instruction dependence, both of which can lead to pipeline stalls.

The memory access irregularities can be addressed by memory hierarchy structures and multi-level caching schemes, which have been anticipated even by von Neumann himself. Each consecutive memory has a different latency-size trade-off, and the optimization approach leverages the different probabilities of memory access, i.e. consecutive memory accesses have a higher probability. The addressing modes with memory reference specifiers can make inter-instruction dependence detection very difficult. This issue is dealt simultaneously at microarchitecture and compiler levels [22, 23]. Notably, CPU performance has been improved over the years by applying a range of optimization techniques: subword parallelism, instruction-level parallelism, cache optimization, thread-level parallelism, etc.

1.2 Programmable logic and Field Programmable Gate Arrays

Another essential technology is the *Field Programmable Gate Arrays* (FPGAs), which have become one of the key mediums for digital circuit implementation. FPGA is a prefabricated silicon device that can be electrically programmed to become almost any kind of digital circuit or system [24]. Simplified FPGA structure is illustrated in Fig. 1.3, it consists of general logic, memory, *Digital Signal Processing* (DSP) blocks, routing fabric and programmable *Input/Output* (I/O) blocks.

Although the first programmable logic became available by the mid-1960s, the first modern-

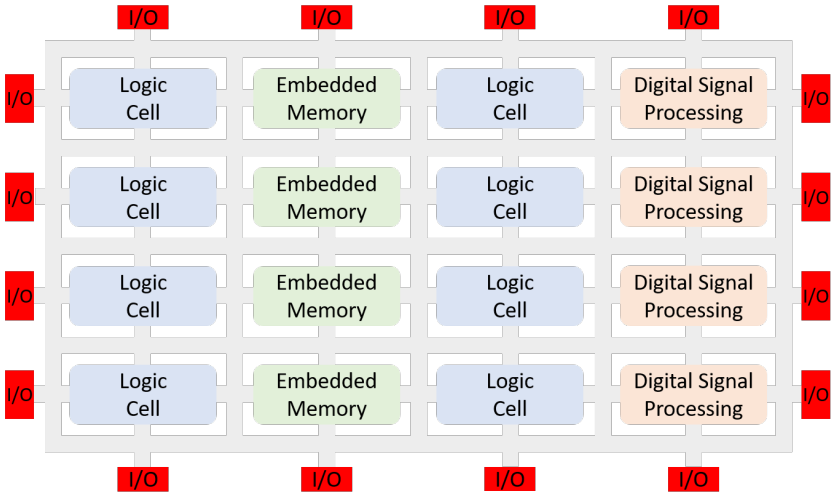


Figure 1.3. Simplified FPGA structure.

era FPGA was introduced by Xilinx in 1984 [24]. The FPGAs are programmed in the ”field” every time upon power-up as opposed to the specialized ICs which are prefabricated in a fab. The programmability and monetary considerations have led to numerous applications of the FPGA technology: compensation for non-existing ICs, prototyping/emulation of *Application Specific Integrated Circuits* (ASICs), low-latency customized communication and data routing, data preprocessing and analysis, HPC, *Deep Neural Networks* (DNNs) and many others.

FPGA development often involves balancing between performance and resource utilization. Unlike processor systems where software directs the execution of an algorithm, in FPGAs, different physical parts of the chip can be dedicated to a specific task. The development process involves the actual design of a digital circuit, which is described using specialized *Hardware Description Languages* (HDLs). Then the described circuit is synthesized into an abstract netlist for the specified technology, which further is fed into placement, routing and programmable file generation processes.

Conventional FPGA development can be categorized into three tracks: 1) synthesis, 2) physical design and 3) verification, as shown in Fig. 1.4. The flow starts with RTL design files which are accompanied by a *testbench*. The testbench provides a virtual experiment for generating stimuli and verifying the anticipated responses. Initially, simulation verifies the functional correction of the circuit description. The synthesis process generates a netlist[7] and a delay file, which permits verification of the synthesized design. Finally, the design is placed and routed

[7]Description of the connectivity of an electronic circuit

for the given FPGA chip. This process produces an FPGA configuration file and delay file for the routed design. Now the placement and routing of the circuit can be verified. Finally, the circuit's operation is confirmed on the physical chip [25]. Usually, the synthesis, placement and routing processes are time-consuming when compared to simulation, therefore it is productive to verify and fix the design as early as possible in the development flow.

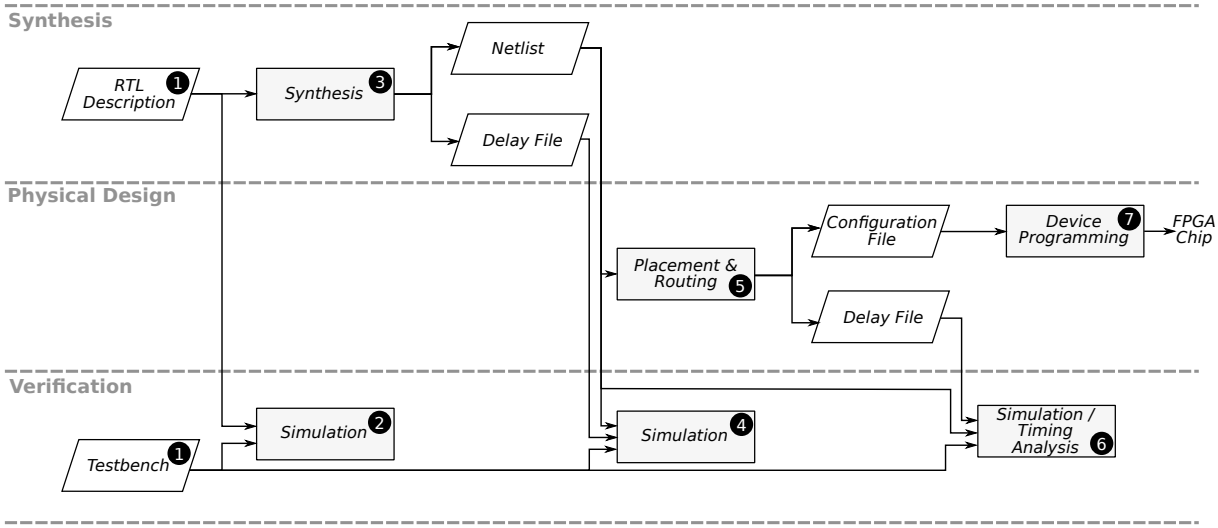
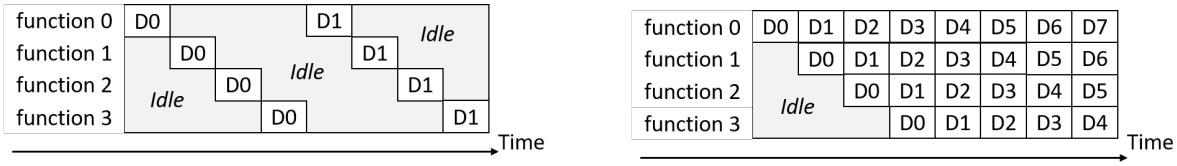


Figure 1.4. RTL development flow of a medium-sized (no partitioning required) design targeting FPGA [25].

FPGA development resembles the design of an ASIC. However, there are additional requirements, i.e. inclusion of testing tracks. The additional testing is necessary to detect defects in the fabrication process, which in the case of FPGAs, is done by the vendors. Furthermore, due to monetary considerations and tape-out delays, testing is an integral part of the ASIC design, which includes self-test circuits, scan registers and the insurance of extensive fault coverage [25].

One of the main performance-increasing techniques for any digital computing system is *pipelining*, which is a special kind of concurrency increasing the system's performance by overlapping the processing of several tasks [25]. A fully pipelined solution can accept new input data on every clock cycle and is characterized by a fixed latency. Examples of non-optimized and optimized processing pipelines are shown in Fig. 1.5a and Fig. 1.5b, respectively. In figures, *functions* represent different hardware blocks while $D0 - D7$ denote data samples.

Although describing customized hardware components in HDL allows designers to adapt existing tools for logic synthesis and to explicitly describe hardware, this approach requires the



(a) Non-optimized processing pipeline **(b)** Optimized processing pipeline

Figure 1.5. Comparison of non-optimized and optimized processing pipelines (The gray area indicates idleness of the functional block).

designer to specify functionality at a low level of abstraction, where cycle-by-cycle behaviour is completely specified [26]. The use of such languages requires advanced hardware expertise and involves cumbersome development, which leads to high time-to-market expenses. Notably, this has led to an increasingly popular design methodology involving the making use of HLS tools, where high-level software language (e.g. C, C++ and SystemC) describes the circuit's functionality. While the tools generate RTL description, the further design flow resembles the one depicted in Fig. 1.4. This high-level approach requires some hardware expertise but can provide maintainability and enable rapid design space exploration [26].

1.3 Other Computational Approaches

Notably, other technological approaches have been proposed and successfully used for high-performance and low-cost processing. These approaches utilize such technological paradigms as GPUs, *Application-Specific Instruction-set Processorss* (ASIPs) and *Coarse-Grained Reconfigurable Arrays* (CGRAs). Although these technologies are not in the focus of the thesis, it is necessary to recognize the spectrum of different computational paradigms to appreciate the challenge of efficient processing and to understand the achieved results in a broader context.

One of the better accepted technologies for scalable computing is GPUs. The GPU provides a much higher instruction throughput and memory bandwidth when compared to the CPU within a similar price and power envelope [27]. While the CPU excels at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU excels at executing thousands of them in parallel [27]. The GPU is specialized for highly parallel computations, and therefore more transistors are devoted to data processing rather than data caching and flow control [27]. A high-level comparison of both architectures is shown in Fig. 1.6.

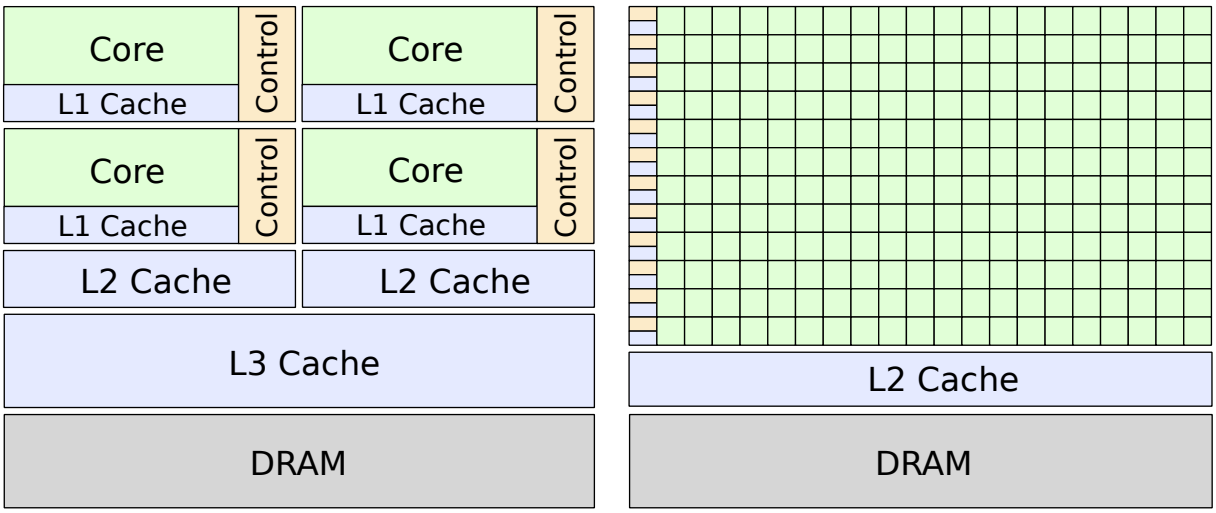


Figure 1.6. High-level comparison of CPU (left) and GPU (right) architectures [27].

The prospect of general-purpose GPU computing has brought the emergence of many programming interfaces. The most popular ones are the proprietary *Compute Unified Device Architecture* (CUDA) interface, which is used exclusively for NVIDIA GPUs, and an *Open Computing Language* (OpenCL) interface applicable for a multitude of technologies, even FPGAs. Both interfaces and GPUs, in general, have been rightfully adopted by the scientific community due to a relatively gradual learning curve, especially for C/C++ programmers [28, 29].

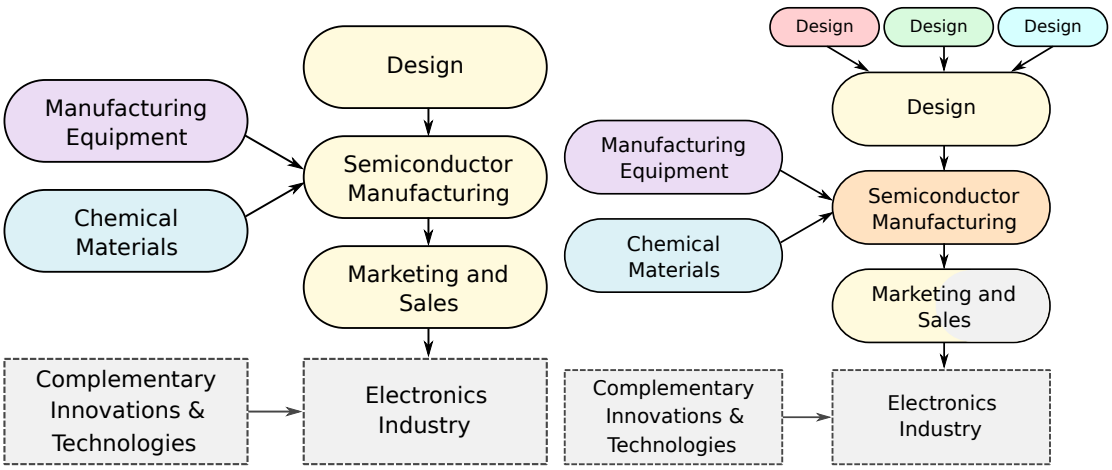
Otherwise, the demand for achieving ASIC-level performance while keeping FPGA-like flexibility has resulted in the emergence of *Coarse-Grained Reconfigurable Array* (CGRA) technology. The CGRAs have domain-specific flexibility where hardware can be defined by software at runtime, but the processing elements are more coarse-grained when compared with FPGAs. CGRAs take advantage of time-multiplexing resources, enabling both temporal and spatial computation and obviating costly deep pipelines and centralized communication. Although CGRA programming models still require improvement, the potential of the technology, for example, is recognized by Intel's project to incorporating them into Xeon processors [30].

Another appealing technology for efficient computing is *Application-Specific Instruction-set Processorss* (ASIPs). ASIPs often can be a part of a larger SoC and is a result of the *Component Specialization* technique [31]. The main difference between a general-purpose processor and an ASIP is the application domain. The design focus of an ASIP is the domain-specific performance and flexibility with a low cost of solving computational problems. The ASIP design process starts with some minimalistic processor core (in the present day, usually RISC-V) which

is profiled against source codes of the domain application(s). Further, the results of profiling are used to select appropriate ISA extensions, perform data path and memory specialization and design tailored accelerated instructions [31, 32]. ASIP design involves multiple disciplines and requires comprehensive knowledge of the target application.

1.4 Heterogeneous System on Chip

Core enablers for reducing energy consumption and enabling personalized mobile computing are *System on Chip* (SoC) and their extension *Heterogeneous System on Chip* (HSoC) technologies. An SoC integrates a range of *Silicon Intellectual Property* (SIP) cores designed by different teams around the world. These SIP cores integrate processors, caching hierarchies, interconnects, co-processors, accelerators, interfacing controllers, memories, encryption engines, peripheral controllers, analogue circuits and many others into a single chip. This integration is carried out using a single- or multi-die design depending on the monetary considerations, i.e. faults introduced by manufacturing are constant across the area, and larger chips are deficient more frequently.



(a) Business model until 1987. **(b)** Modern business model.

Figure 1.7. Comparison of traditional (before 1987) and modern business models of semiconductor companies [33]. Colours represent separate companies.

The SoC industry relies on the reuse and adaptability of the SIPs cores, facilitated by the evolving IC business models and standardization of on-chip communication protocols. The first foundry - *Taiwan Semiconductor Manufacturing Company* (TSMC) - was established in 1987, utilizing a unique business model, which boosted the collective productivity of the semicon-

ductor industry as new companies without any manufacturing capabilities now could develop profitable and innovative ICs. Nevertheless, the IC design phase usually was localized to a single team working in a single company. It was changed by one of the first fabless SIP companies formerly based in the UK - *Acorn RISC Machine* (ARM) where RISC - *Reduced Instruction Set Computer*. ARM decided to focus on the designs, i.e. SIPs, instead of marketing and selling physical chips. Currently, ARM designs the most widespread processors, integrated into mobile phones, tablets, vehicles, embedded systems, etc. At the time, other companies were attempting this business model to work, but the "revolution" brought in by ARM was the definition and publication of unified communication protocols - *Advanced eXtensible Interface* (AXI) *Advanced Microcontroller Bus Architecture* (AMBA) specification, which enabled interchangeability and massive re-usability of the designed SIPs. At this point, any chip designer could assemble their chip from a catalogue of modular verified IP cores, each developed by an experienced team [33].

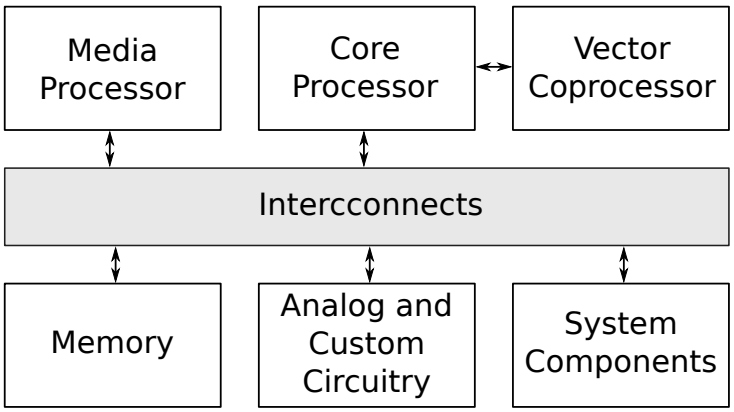


Figure 1.8. A simplified SoC system model [34].

Fig. 1.8 shows a classical high-level structure of an SoC. The system usually incorporates a single or multi-core processing unit (often ARM-based), an external memory interface, system-level interconnections and system components, e.g. for interfacing, data movement and encryption. The SoC may also include additional domain-specific processors and circuitry depending on the use case. The system-level interconnection is central to the performance and reliability of the SoC, and it can be classified into two categories: bus-based (such as AMBA) and *Network on Chip* (NoC) approaches. The bus-based interconnects usually are comprised of two or more hierarchical sub-buses to optimize system-level performance and costs, i.e. the bus closest to the CPU has the highest bandwidth. The NoC approaches utilize a mesh of standardized elements implementing a packet-based communication scheme. Apart from processors, the

memory interface usually is also utilized by *Direct Memory Access* (DMA) engines and high-speed controllers, such as *Universal Serial Bus* (USB), Ethernet and PCIe [34]. An example of a commercial NXP i.MX 6 SoC architecture is provided in Appendix 1.

With the increase of connectivity and data production, there is a growing need for new processing capabilities and more performant solutions. While different computing paradigms (CPU, FPGA, GPU, DSP) excel at different processing challenges, the best mixture of performance, efficiency and costs require a complex utilization of these computing capabilities. This has led to the fabrication of HSoCs, where these different abilities are combined within a single chip. Although in this thesis we mainly utilize an FPGA-based HSoCs, the same principles of algorithm partitioning apply to other devices.

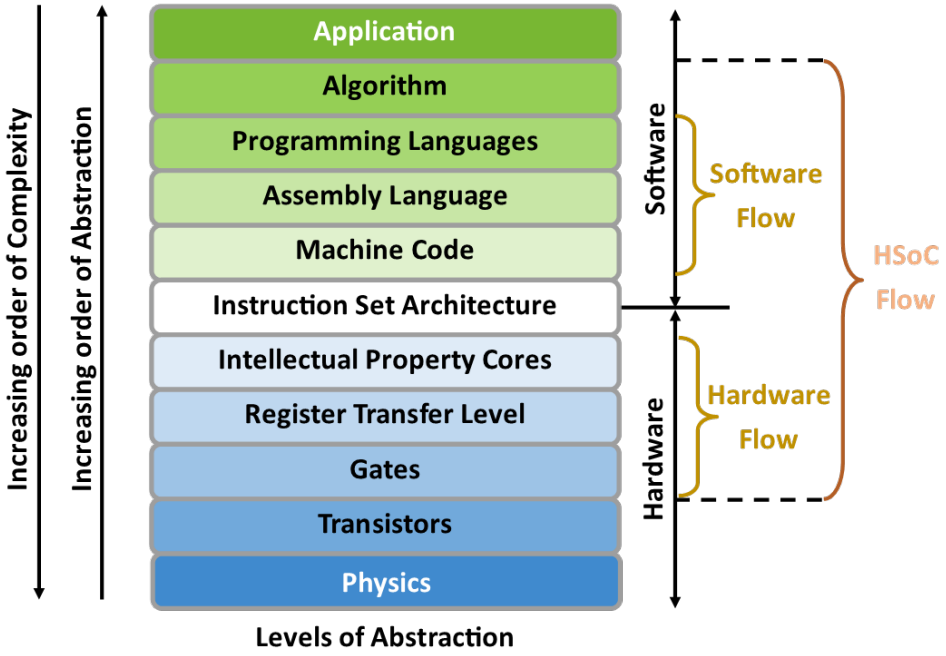


Figure 1.9. Levels of abstraction for an electronic computing system.

The simultaneous utilization of diverse computing paradigms presents the challenge of harmonizing different design flows. Fig. 1.9 represents a variety of abstraction levels that are a part of any semiconductor-based application. Typically, ISA manages the interface between hardware and software. Therefore, in classical software development, everything is fixed below and including the ISA abstraction level. Analogously, the classical FPGA development flow is not concerned with software. Classically, when the computing system incorporates both the sequential processing and digital design paradigms, the design team is split in two, and the system

architect defines the software/hardware interfacing effectively breaking up the development into parallel design flows. Otherwise, the degree of HSoC integration permits the development of more customized hardware-software solutions. These solutions require multi-disciplinary expertise concerning not only implementation but also the algorithm that impacts all stages of the development and calls for the consideration of such factors as arithmetic precision of the accelerator, locality and pipelining potential of the algorithm, choice for on-chip data movement, the robustness of the kernel driver implementation, efficiency of kernel/user API, real-time characteristics of the system, etc. An example of a relatively complex HSoC incorporating FPGA, GPU, multiple processors and a wide range of SIPs is provided in Appendix 4.

1.5 Digital interfaces

Advancement in chip manufacturing technology is a major factor ensuring the availability of SoC technology, but another factor is the standardization of the communication interfaces. Currently, ARM's ISA is the most widespread among SoCs and the reason for that is the development and outspread of standardized on-chip communication protocol - AMBA. The standardization has been a tremendous push to the re-usability and compatibility, as from now on, different parties across the globe were able to develop their designs following this specification and assure compatibility with ARM-based systems. From the perspective of the SoC vendor, this presented a list of silicon IP cores, which were available for licensing and immediate implementation, thus reducing time-to-market expenses. In the context of HSoCs, the FPGA logic interacts with the exposed interfaces of the hard[8] processing system.

Any interface acts as a set of communication rules between a *master* and *slave* devices as shown in Fig. 1.10. In some specifications, the master provides a *source* of an interface, while the slave provides a *sink*. Although usually interfaces are considered to have a certain degree of complexity, the abstraction of an interface can be (and often is) applied to a single wire or signal, e.g. clock, reset or interrupt. Some higher-level interfaces may be classified by protocol type, e.g. memory-mapped, streaming or conduit.

Some of the aspects of the thesis require distinguishing between memory-mapped and streamed interfaces. Memory-mapped interfaces usually govern the majority of the chip's inner (inter-SIP) communications. The main identifying characteristic of such an interface is its address

[8] As opposed to fabricated in FPGA's (soft) logic.

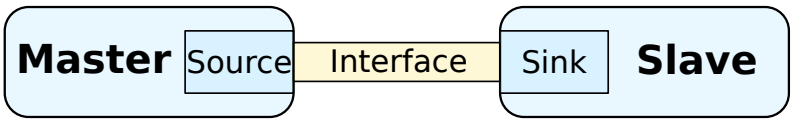


Figure 1.10. Simplified view of interface between Master and Slave.

signal that enables an effective routing of the memory-mapped request by internal interconnects and other circuitry, e.g. *Memory Management Unit* (MMU), cache controller. The Fig. 1.11 shows a simplified usage example of a memory-mapped interface and an example specification of the *Avalon Memory-Mapped* interface [35] is provided in Appendix 5. Note that many signals are optional, and their choice is dependant on the needs of the actual hardware.

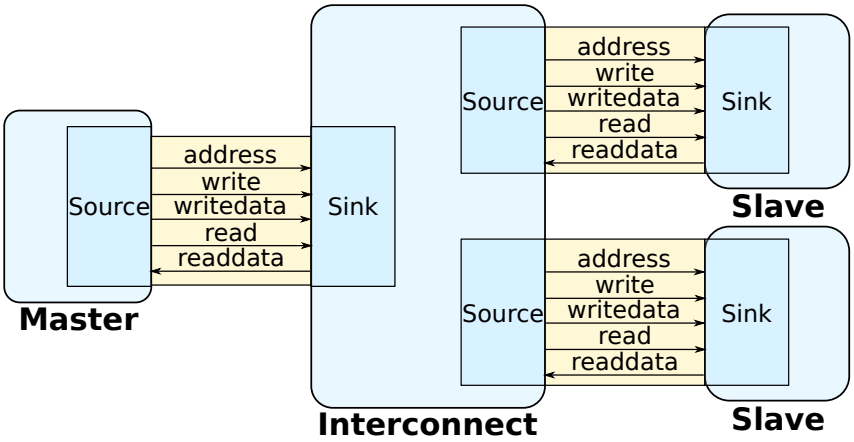


Figure 1.11. A simplified usage example of a memory-mapped interface.

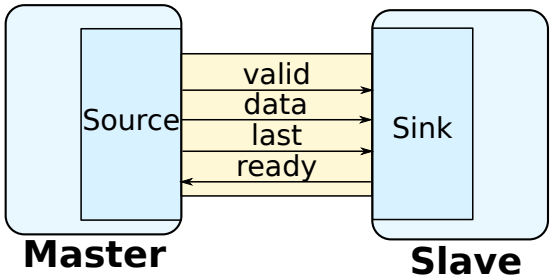


Figure 1.12. A simplified usage example of streaming interface.

Another essential communication type is the *streaming* interface that omits the address signal and usually is present in computing pipelines. The streaming interface usually provides a handshake mechanism to exchange information on the validity of the data and the receiver readiness. There are other optional signals for carrying a range of information, e.g. start and stop of the transfer frame, byte validity of multi-byte data bus, user's sideband signals without any strict definitions, etc.

1.6 Linux operating system

When encountered with the limitations of the contemporary low-cost, UNIX-based Minix OS Linus Torvalds famously decided to write his operating system in 1991. The use of Linux took off and, because of its license, became a collaborative project developed by many [36]. Linux has become very influential and is used by systems of varying scope, ranging from large-scale servers to modern embedded systems [37]. Linux has considerably low requirements, supports a wide range of customization, provides multi-threading and has a variety of available open-source software which can speed up development.

OS is responsible for the elementary use and management of resources available to the system, i.e. processor execution time, memory, storage elements, connected devices, network interfaces, etc. The primary management technique for achieving this goal is virtualization, where the OS takes a physical device, such as the processor or memory and transforms it into a more general, easy-to-use virtual form. This virtualization is provided to the user applications via a standardized system call interface[9] as shown in 1.13.

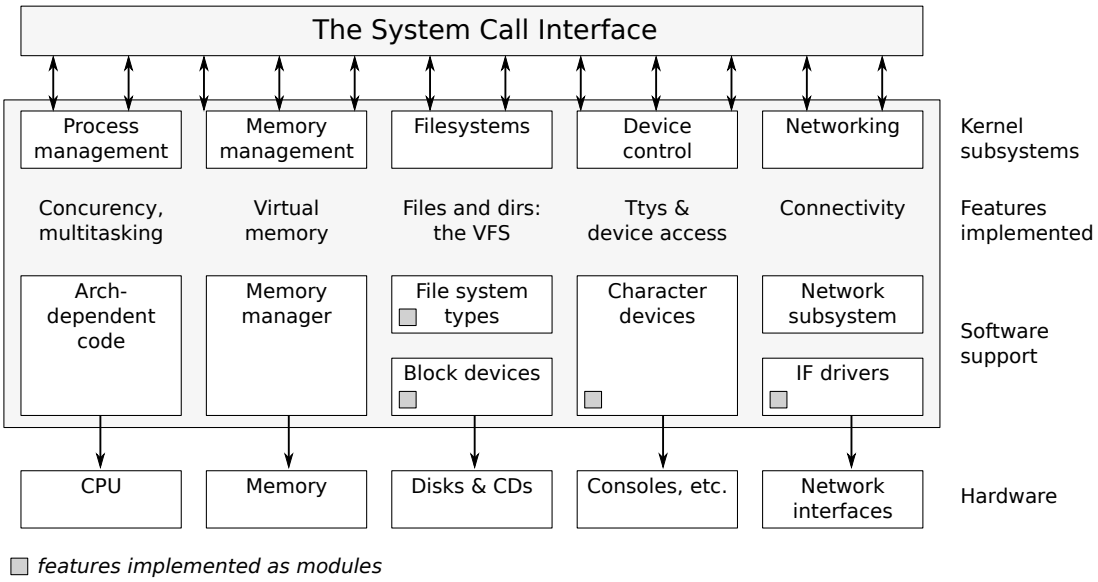


Figure 1.13. Basic structure of the Linux kernel and system call interface [38].

One of the most critical considerations for HSoC-based designs is the concept of virtual memory, which isolates applications, provides means for efficient inter-process communication, enables requesting more dynamic memory than there is available (via page-swapping) and

[9]Notably, Linux provides an alternative popularity-gaining kernel interface mainly used for networking applications - *Berkeley Packet Filter* (BPF)

improves security in general [38]. The processor executes using virtual address space that is translated to a physical address space by the MMU. Both the virtual memory and the physical memory are broken into pages so that a virtual page is mapped to a physical page [23]. Each processing core has its own MMU, which also acts as an independent system bus master. The CPU-MMU relationship and an example of a two-stage physical address lookup for a 32-bit system using 4 KB pages is shown in Fig. 1.14. The translation tables reside in the system memory, and MMU is configured with the physical base address of the first-stage translation table. For performance purposes MMU stores translations in a *Translation Lookaside Buffer* (TLB). In addition , mappings hold flags that determine access and caching policies, e.g. device configuration space must not be cached. The kernel manages these translation tables for all processes and updates MMU lookup root address upon every context switch. Generally, processes share kernelspace translations while userspace mappings are shared between threads or as a part of a shared memory *Inter-Process Communication* (IPC) mechanism. If, for example, the user application produces invalid memory access, the MMU generates a page fault for handling by the kernel.

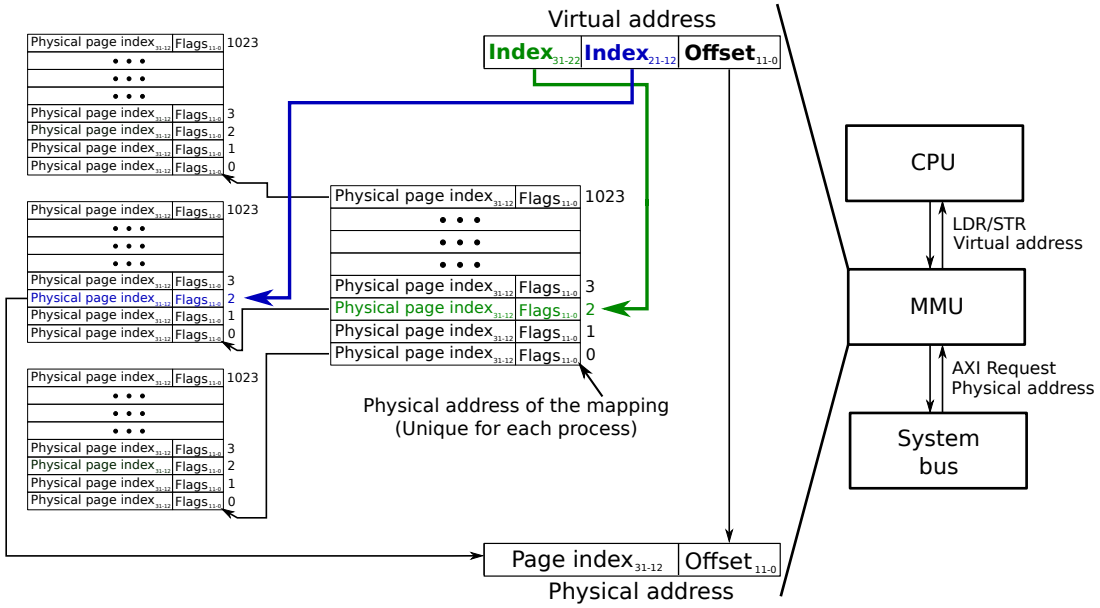


Figure 1.14. An example of 4 KB page-based virtual memory management in a 32-bit ARM-based.

Page-based memory virtualization ensures that physically non-contiguous memory appears contiguous for software as shown in Fig. 1.15. Nevertheless, other bus masters (apart from

CPU) operate using physical address space[10]. While some masters (such as PCIe) can reserve memory at startup, it still presents a challenge for HSoC-based system development when both CPU and FPGA share access to the same system memory. A popular option is to utilize scatter-gather DMA that requires constructing a linked list of transaction descriptors, a process that can be inefficient for large transfers. An alternative is to allocate a large region of contiguous memory region and disable its swapping by using specialized features in the OS, e.g. Linux contiguous memory allocator (introduced in 2012 [39]).

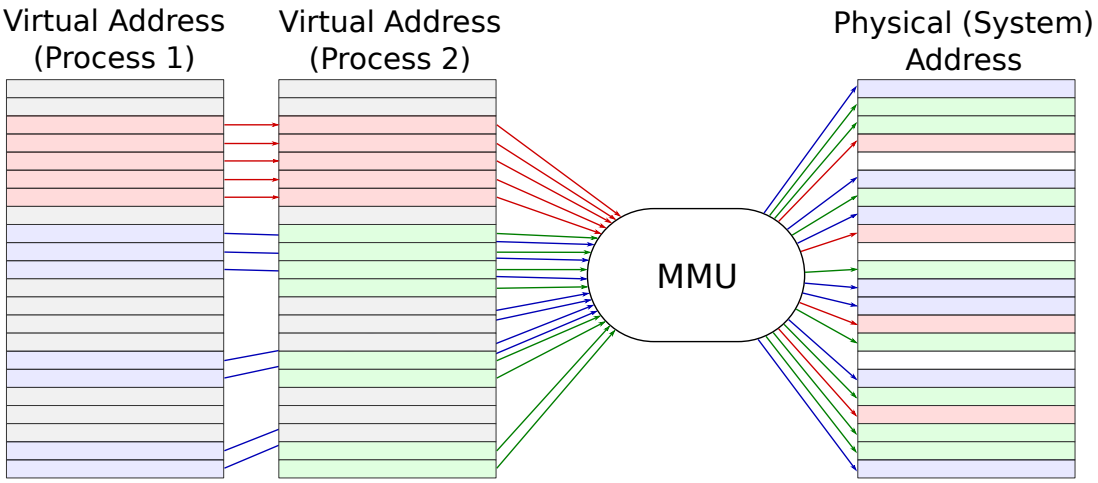


Figure 1.15. Simplified example of page-based virtual-physical address space mapping for two processes. Pages illustrated in red denote kernelspace while blue and green pages denote two distinct applications.

A complete HSoC-based system design also implies kernel-level development because kernelspace accounts for communication between software and designed hardware (accelerators) and implements the interface for the userspace. Importantly, there is a distinction between userspace and kernelspace accompanied by different access permissions. To properly interface the custom hardware accelerator with the actual application in the context of HSoC technology, one has to touch upon the whole level of software abstractions: ranging from the development of the device driver to the application.

[10]This statement does not hold for systems with *Input/Output Memory Management Unit* (IOMMU).

2. COMPUTER VISION

This section lays out the core principles of the general projective camera model and the adopted mathematical notations necessary for the comprehension of contributions depicted in Section 4. The section also presents such relevant concepts as lens distortions, epipolar geometry, feature extraction and stereo correspondence calculation. Additionally, considerations for different stereo correspondence methods are aggregated and analysed. Finally, the section outlines the relevant technicalities of modern *Machine Learning* (ML)-based algorithms.

2.1 General Projective Camera

In principle [40], a camera is a mapping between the 3D world and a 2D image. Modelling of all cameras is based on a notion of a *general projective camera*, which can be conveyed as a matrix **P** which maps world points Q to the image points q.

Let's consider plane at $\mathrm{Z} = \mathrm{f}$, which is called *image plane* or *focal plane*. The pinhole camera model determines that a point in space $\mathrm{Q} = (\mathrm{X}, \mathrm{Y}, \mathrm{Z})^T$ is mapped to the point on the image plane where a line from Q to the centre of the projection meets the image plane, as shown in Fig. 2.1.

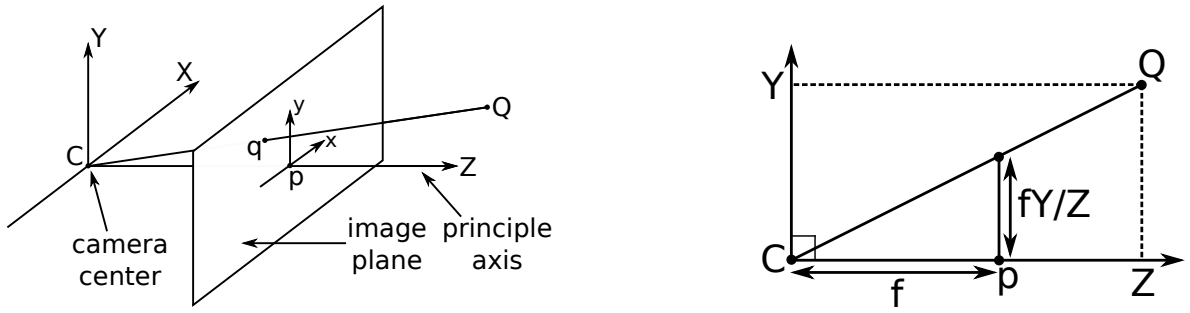


Figure 2.1. Pinhole camera geometry (image plane is mirrored towards the scene).

From similar triangles, it can be shown that $(\mathrm{X}, \mathrm{Y}, \mathrm{Z})^T$ maps to $(\mathrm{fX}/\mathrm{Z}, \mathrm{fY}/\mathrm{Z}, \mathrm{f})^T$. If the world and image points are represented in homogeneous coordinates, then point projection can be expressed in terms of matrix multiplication as:

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{y} \\ \omega \end{pmatrix} = \begin{pmatrix} \mathrm{f} & 0 & 0 & 0 \\ 0 & \mathrm{f} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \\ 1 \end{pmatrix}. \tag{2.1}$$

The equation can be rewritten in a matrix form as:

$$\mathrm{q} = \mathbf{P}\mathrm{Q}, \tag{2.2}$$

where **P** denotes *camera projection matrix*.

The Eq. 2.1 assumes that the origin of the coordinates is in the image plane at the principle point. Usually, this is not the case as the origin of coordinates for digital images starts at the corner that can be accounted for simply by adding an offset as shown in Eq. 2.3.

$$\begin{pmatrix} x \\ y \\ \omega \end{pmatrix} = \begin{pmatrix} \mathrm{f} & 0 & p_x & 0 \\ 0 & \mathrm{f} & p_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}. \tag{2.3}$$

Often the matrix is rewritten by denoting matrix K, which is called the *camera calibration matrix* [40], as:

$$\mathbf{K} = \begin{bmatrix} \mathrm{f} & 0 & p_x \\ 0 & \mathrm{f} & p_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.4}$$

thus the description of the projection reduces to its concise form:

$$\mathrm{q} = \mathbf{K}[\mathrm{I}|\mathrm{O}]\mathrm{Q}. \tag{2.5}$$

Previously we have used a coordinate system which may be called the *camera coordinate frame*. In general, the points in space can be expressed in the Euclidean coordinate frame, otherwise referred to as the *world coordinate frame*, this is of great importance for virtual environments, more specifically synthesizing virtual cameras. Both coordinate frames are related through rotation and translation operations as shown in Fig. 2.2.

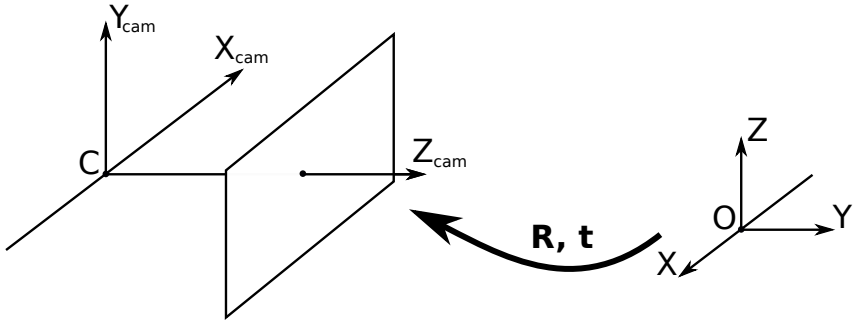


Figure 2.2. Relationship between camera and world frames [40].

Let $\widetilde{\mathrm{Q}}$ represent an in-homogeneous coordinate vector of a point in the world coordinate frame, and Q represent the same point in the camera coordinate frame. Then we may write that

$\mathrm{Q} = \mathbf{R}(\widetilde{\mathrm{Q}} - \widetilde{\mathrm{C}})$, where $\widetilde{\mathrm{C}}$ represents the coordinates of the camera in the world frame, and $\mathbf{R}$ is a 3×3 rotation matrix representing the orientation of the camera coordinate frame. The equation can be expressed in homogeneous coordinates as:

$$\mathrm{Q} = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\widetilde{\mathrm{C}} \\ 0 & 1 \end{bmatrix} \begin{pmatrix} \widetilde{\mathrm{X}} \\ \widetilde{\mathrm{Y}} \\ \widetilde{\mathrm{Z}} \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\widetilde{\mathrm{C}} \\ 0 & 1 \end{bmatrix} \widetilde{\mathrm{Q}}. \tag{2.6}$$

Inserting in the Eq. 2.5 gives:

$$\mathrm{q} = \mathbf{KR}[\mathrm{I}| - \widetilde{\mathrm{C}}]\widetilde{\mathrm{Q}}, \tag{2.7}$$

where $\widetilde{\mathrm{Q}}$ now represents a point in a world coordinate frame. Now it can be seen that the general pinhole camera, $\mathbf{P} = \mathbf{KR}[I| - \widetilde{\mathrm{C}}]$, has 9 degrees of freedom. 3 for $\mathbf{K}$, 3 for $\mathbf{R}$ and 3 for $\widetilde{\mathbf{C}}$. The parameters contained in the $\mathbf{K}$ are called the *internal* parameters, or the *internal orientation* of the camera, while the parameters $\mathbf{R}$ and $\widetilde{\mathbf{C}}$ are referred to as *external* parameters or the *exterior orientation* [40].

Thus far, we have assumed that the image coordinates are in the Euclidean coordinate frame and have equal scale in both axial directions. In the case of many optical systems and *Charge-Coupled Device* (CCD) cameras, there is the additional possibility of having non-square pixels, thus introducing unequal scale factors in each direction. In particular, let m_x and m_y be the number of pixels per unit distance in image coordinates in x and y directions, then the transformation from world coordinates to the pixel coordinates is obtained by multiplying Eq. 2.4 on the left by an extra diagonal factor $(m_x, m_y, 1)$, thus the general form of the calibration matrix becomes:

$$\mathbf{K} = \begin{bmatrix} \alpha_x & 0 & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.8}$$

where $\alpha_x = \mathrm{f}m_x$ and $\alpha_y = \mathrm{f}m_y$ represent the focal length of the camera in terms of pixel dimensions in the x and y directions respectively. Similarly, (x_0, y_0) is the principal point in terms of pixel dimensions, with coordinates $x_0 = p_x m_x$ and $y_0 = p_y m_y$.

Even more general camera matrix may add a *skew* parameter s, thus the camera calibration matrix becomes

$$\mathbf{K} = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.9}$$

However, for most normal cameras, skew parameter can be assumed zero.

2.2 Lens Distortions

Image processing models assume that cameras obey a *linear* projection model where a straight line in the world results in a straight line in the image [41]. Unfortunately, lenses, which focus light onto the camera pixel matrix, have noticeable lens distortions that manifest as a visible curvature in the projection of straight lines, illustrated in 2.3.

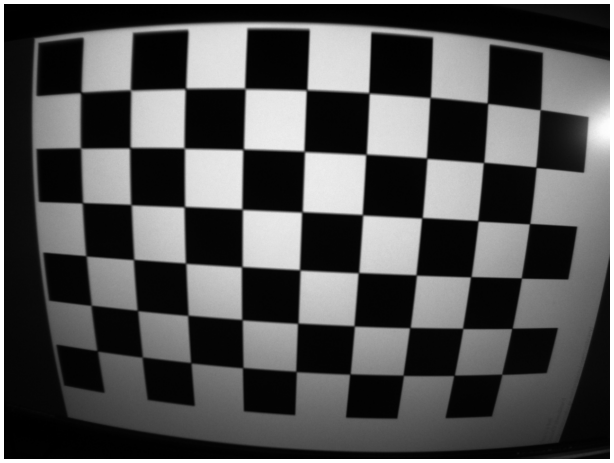


Figure 2.3. An example of lens distortion effect on the calibration image for bumblebee stereo camera.

Unless these distortions are corrected, it is impossible to create highly accurate photo-realistic reconstructions [41]. Fortunately, for most of the lenses, these distortions can be corrected using a simple quadratic model [42], which produces good results. The implementation of such algorithms in the digital logic is much more involved and is discussed in Section *4.3.3 An approach to spatial image transformation*. The image distortion takes place during the initial projection of the world onto the image plane; thus, *calibration matrix* reflects a choice of affine coordinates in the image, translating physical locations in the image plane to pixel coordinates [40]. Let $(\widetilde{\mathrm{x}}, \widetilde{\mathrm{y}})$ be the non-distorted pinhole projection. The radial (lens) distortions can be modelled as:

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = L(\widetilde{r}) \begin{pmatrix} \widetilde{x} \\ \widetilde{y} \end{pmatrix}, \tag{2.10}$$

where

- $(\widetilde{\mathrm{x}}, \widetilde{\mathrm{y}})$ is the ideal image position (which obeys linear projection),
- $(\mathrm{x_d}, \mathrm{y_d})$ is the actual image position, after radial distortion,
- $\widetilde{\mathrm{r}}$ is the radial distance $\sqrt{\widetilde{x^2} + \widetilde{y^2}}$ from the centre for radial distortion,
- $\mathrm{L}(\widetilde{\mathrm{r}})$ is a distortion factor, which is a function of the radius $\widetilde{r}$ only.

In pixel coordinates the correction can be rewritten as:

$$\begin{aligned} \widetilde{x} &= x_c + (x - x_c)L(\widetilde{r}) \\ \widetilde{y} &= y_c + (y - y_c)L(\widetilde{r}) \end{aligned}, \tag{2.11}$$

where (x, y) are the "required" coordinates of the corrected image, and $(\widetilde{x}, \widetilde{y})$ are the coordinates in the distorted (input) image.

The distortion factor is an arbitrary function defined for positive values and at the center of the distortion $L(0)$ is 1. Distortion factor can be given by a Taylor expansion

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + ..., \tag{2.12}$$

where the coefficients for radial correction $k_1, k_2, k_3, ...$ are considered a part of the interior calibration of the camera [40]. Notably, the polynomial model can achieve a good approximation for less distorted lenses by using only two coefficients; nonetheless, the high-order model can significantly increase computational complexity. Therefore other models exist that are not a part of the developed image processing pipeline described further, e.g. division model [43].

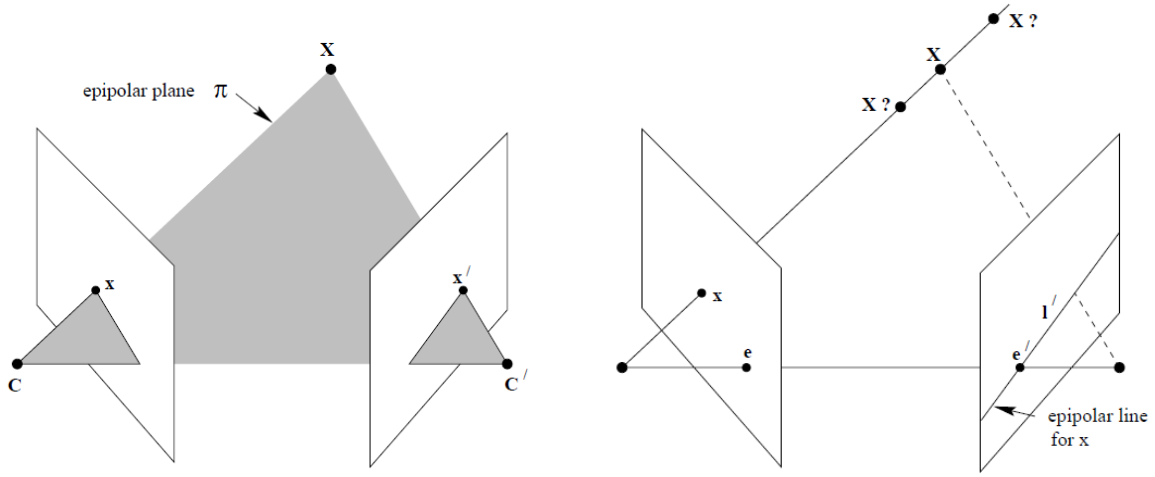
2.3 Epipolar Geometry

An important part of any use case involving 3D reconstruction from images involves epipolar geometry that relates two views based only on the cameras' internal parameters and relative pose. This relationship is encapsulated by a *fundamental matrix* **F**. **F** is a 3×3 matrix, and it has such property that if the point in 3D space X projects as x in the first view and as x' in the second view, then the following relationship is satisfied:

$$x' \mathbf{F} x = 0. \tag{2.13}$$

The fundamental matrix is independent of scene structure; however, it can be computed from the correspondences of imaged scene points without requiring knowledge of the cameras' internal parameters or relative pose.

The epipolar geometry between two views essentially describes the geometry of intersection of the image planes with the pencil of planes having the baseline as an axis [40]. The epipolar geometry is motivated by the search for correspondences across the views that lay on a plane in 3D space and corresponds to the lines in the images, called *epipolar lines*, i.e. consider the images in Fig. 2.4.

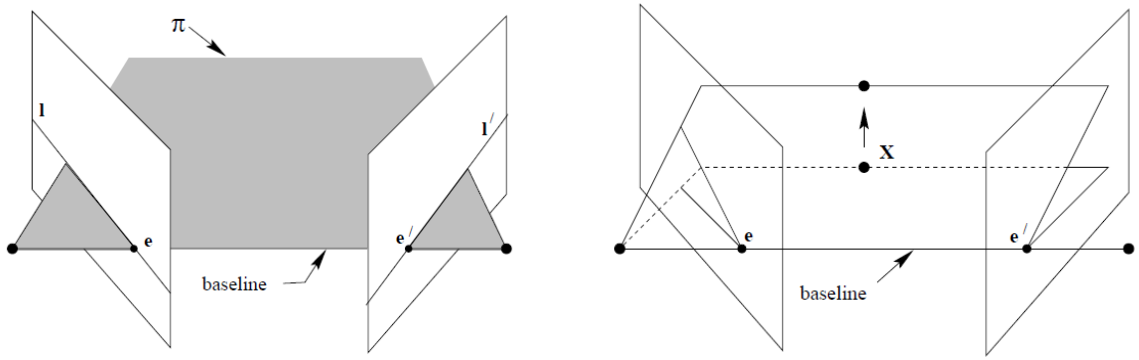


(a) Plane constructed by a 3D point corresponding projections in the views.

(b) A point in one image is anticipated to be located at a line in the other one.

Figure 2.4. Point correspondence geometry [40].

The image points x, x', space point X, and camera centres are coplanar. Let's denote this plane as π. The rays back-projected from the x and x' intersect at X are coplanar, lying in π. This is well illustrated in 2.4b, where the same point in one image corresponds to multiple points in the other image, but still, all these corresponding rays lay in the same plane π.



(a) Camera baseline intersecting the image planes at the epipoles.

(b) Family of planes obtained by rotating the epipolar plane around the baseline.

Figure 2.5. Epipolar geometry [40].

When the x is known, the epipolar geometry constrains x' to the plane π, thus localizing the possible correspondences. The possible correspondences lay on a cross-section between π and image planes, this cross-section is referred to as *epipolar line*. The point of intersection of the line joining camera centres and all of the epipolar lines is called the *epipole*, this is illustrated in Fig. 2.5. Fig. 2.5b shows the rotation of the π plane, thus resulting in the rotation of the

cross-sections and epipolar lines as well.

Fig. 2.6 shows an example of matched points and their corresponding epipolar lines. By following each of the epipolar lines, it can be observed that they cover the same points in both images.

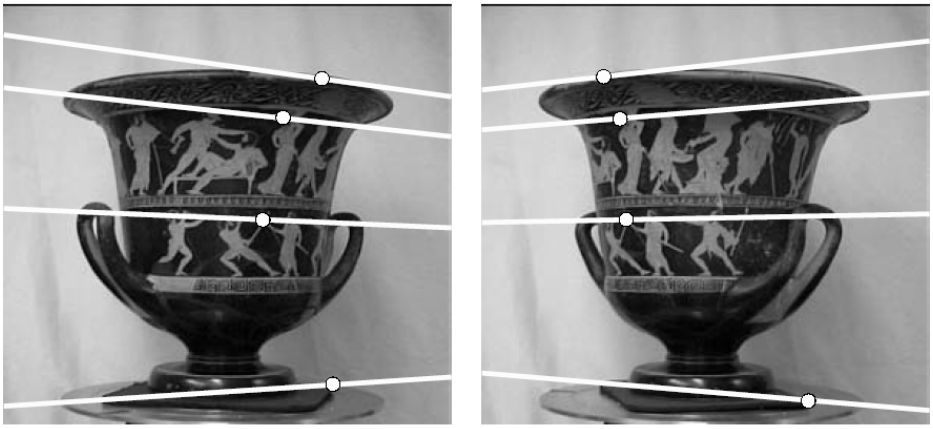


Figure 2.6. An example of stereo image correspondences and their respective epipolar lines [40].

These epipolar lines can be ”straightened” by applying image rectification. Rectification is a process of resampling pairs of stereo images to produce a pair of *matched epipolar projections*, where epipolar lines run parallel with the x-axis; thus, the matching procedure can be performed for consequent disparities between the images in the x-direction only, essentially omitting y disparities.

A pair of 2D projective transformations are applied to the two images in order to match the epipolar lines. Both transformations may be chosen in such a way that matching points have approximately the same x-coordinate. In this way, the two images, if overlaid on top of each other, will correspond as far as possible, and any disparities will be parallel to the x-axis. In effect, two image transformation by the appropriate projective transformations reduces the problem to the epipolar geometry produced by a pair of identical cameras placed side by side with their principal axes parallel. Many stereo matching algorithms assume this geometry, after such rectification the search for matching points is vastly simplified; therefore, that is a preliminary step to comprehensive image matching [40].

2.4 Stereo Correspondence

Stereo correspondence has been and still continues to be one of the most heavily investigated topics in computer science. The term *disparity* was first introduced in the human vision literature [44] to describe the difference in location of corresponding features seen by the left and right eyes.

Early stereo-matching algoithms were feature-based [41], i.e. a set of potentially matchable features are extracted from the images and then searched for the corresponding locations. One of the advantages of such methods is that, images may have different illumination, where edges might be the only stable features [45]. Such local methods are well suited for implementation in digital circuitry due to their localality. More recent methods are focused on first extracting highly reliable features and then using *seeds* to grow additional matches. Similar approaches have also been extended to wide baseline multi-view stereo problems and combined with 3D surface reconstruction [41].

There are two broad classes of stereo algorithms *local* and *global*. In the *local* (window-based) algorithms the disparity computation at a given point depends only on intensity values within a finite window, while the *global* ones make an explicit smoothness assumptions and then solve an optimization problem. In between of these classes there are certain iterative algorithms that do not explicitly state a global minimization function.

Szeliski [40] points out the subset of the following four algorithmic ”building blocks” from which a large set of algorithms can be constructed, this classification is illustrated in Fig. 2.7. This arrangement be used to classify stereo correspondence algorithms even further [46].

Some local methods utilize a support region, essentially combining steps 1 and 2. The global methods make explicit smoothness assumptions that are used for minimizing a global cost function. Another subset of approaches is iterative algorithms, which are based on an image pyramid, where results from coarser levels are used for constraining more local searches.

The solution of the correspondence problem necessitates establishing a metric to compare individual pixels of the stereo image pair. The metrics are based on either grayscale values of the pixels or post-processed features while varying in their ability to expose pixel uniqueness, receptiveness to camera gain or bias, and their computational complexity. A natural measure of similarity is the calculation of cross-correlation [47]. The most popular pixel-based matching

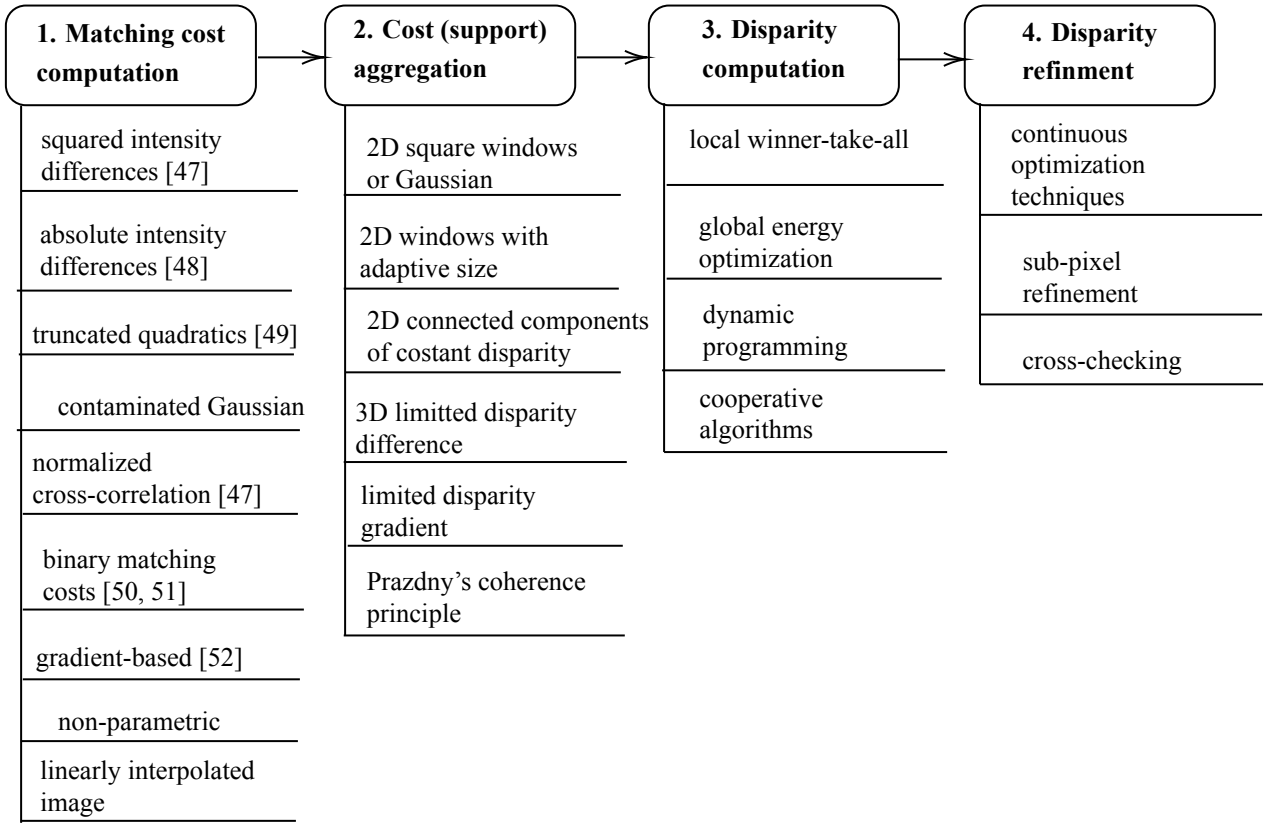


Figure 2.7. Classification of most common steps in disparity calculation

costs include squared intensity differences and absolute intensity differences.

Such costs as census transform and rank transform, are not only insensitive to the camera gain and bias differences but are good candidates for implementation in digital logic. The drawback of such algorithms is their heightened sensitivity to noise. Depending on the depth of disparity calculations (number of pixels across the epipolar line, which are considered for matching), the number and type of operations can be implausible for software or hardware implementation.

Aggregation is done by summing all matching costs over square windows with a constant disparity. Nonetheless, several methods like truncating quadratics and contaminated Gaussians limit the influence of mismatches during the aggregation [53, 54]. In local methods, the emphasis is on the matching cost computation and cost aggregation steps. Computing the final disparities is trivial: simply the disparity is associates pixel with the minimum aggregation cost value, i.e. essentially performing a local *Winner-Take-All* (WTA). The weakness of such methods is that the uniqueness of matches is only enforced for one image, i.e. the *reference image*,

while points in the other image might match multiple points [41].

Notably, many algorithms adopt sub-pixel refinement stages after the initial discrete correspondence stage or alternatively start with more discrete disparity levels [55].

2.5 AI-based algorithms

2.5.1 Technical Background

Since 2012 when the ImageNet Image Classification competition was assuredly won by Krizhevsky, Sutskever and Hinton with their deep-learning-based solution [56], it became evident that *Deep Learning* (DL) algorithms bear the potential for a variety of applications. With the increasing availability of computational resources, ML has become a widely used technique for solving a variety of different problems, e.g. object identification, cluster classification, pattern recognition, functional regression, etc. [57]. Ever since DNNs demonstrated their superior performance, they have been considered for a wide range of use-cases and processing architectures. Considerable effort has been devoted to improve computational efficiency by developing new *Artificial Neural Network* (ANN) architectures [58, 59] and optimizing implementations for specific use-cases [60–62]. The NNs show SoA performance in the stereo-vision competitions, therefore it is necessary to explore realization in programmable logic.

A NNsis a system that is designed to model how the brain performs a particular task or function of interest [63]. A network can be split into fundamental information-processing units - neurons, which form the basis for designing artificial neural networks. The block diagram in Fig. 2.8 shows neuron's mathematical model. Neuron's inputs x_i are multiplied by coefficients w_{ki}, referred to as weights and summed up together with a bias b_k. This sum is passed to an activation function φ, which is used to normalize neuron's output, k and i designates a specific neuron and its input.

A type of frequently used neural network is constructed by arranging neurons in layers where all neurons in every layer are connected to each neuron in the adjacent forward layer, i.e. fully connected feed-forward network. This type of network is illustrated in Fig. 2.9.

By utilizing the neural network's structure it is possible to derive a number of fundamental operations for each layer depending on its input and output count. Let N_{in}, N_{out}, N_{add}, N_{mul}, N_{act} be the number of inputs, outputs, adders, multipliers and activation function operations

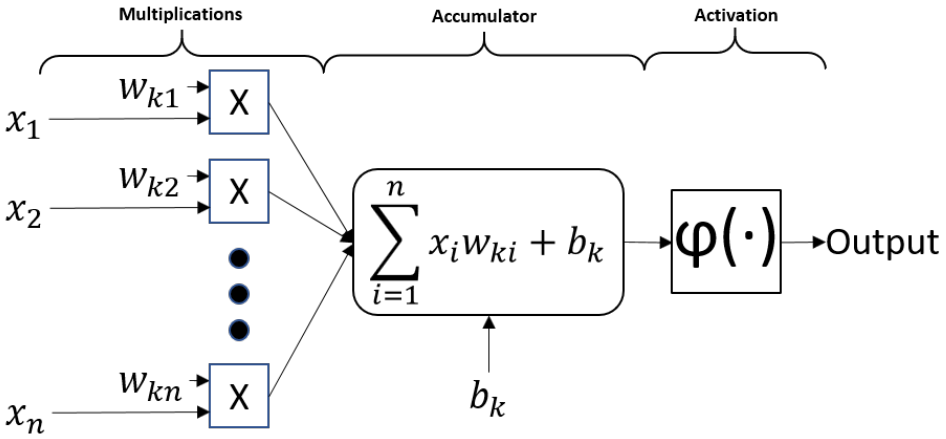


Figure 2.8. Structure of a single neuron.

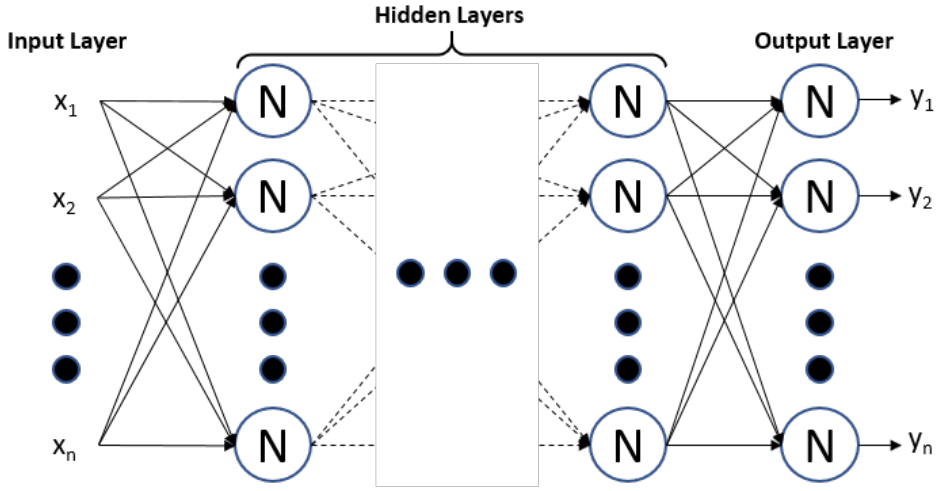


Figure 2.9. General structure of a feed forward neural network.

respectively. The number of multipliers and adders corresponding to a fully pipelined implementation for a neuron is

$$N_{mul} = N_{add} = N_{in}. \tag{2.14}$$

Therefore for each layer it is:

$$N_{lmul} = N_{ladd} = N_{in} \times N_{out}. \tag{2.15}$$

Every neuron has one activation function, thus, for a layer, the theoretical number of required activation function calculations amount to output count.

The notions presented here are further explored in the subsection 4.1.

2.5.2 Related work

Ever since the early formalization of NNs [64], a significant effort has been made and various paradigms used to adopt different NN structures for digital circuit implementation. For example, *Convolutional Neuron Networkss* (CNNs) are widely used for image recognition and classification, although classification itself is carried out by a fully connected FFNN. Different kinds of paradigms ranging from co-processor systems [65] to OpenCL-based solutions [66–70] have been used to find the optimal trade-off between resource use, latency, and throughput. This section highlights relevant aspects of previous FFNN implementations and summarizes the main design challenges.

An implementation of an FFNN using VHDL is presented in [71]. The proposed network is based on a collection of simple-interconnected processing elements (neurons), which are organized into a topology composed of individual layers. The neurons between layers can communicate concurrently. Network's coefficients are represented using one's complement signed fixed-point binary numbers. Different hardware optimization techniques have been suggested, i.e., weight storage in internal memory, Booth's multiplication algorithm and activation function's bilinear approximation using a counter and shift registers. The authors implemented a simple FFNN with a 2-2-1 topology. Network's pipelined performance is estimated to be 34 ns per single output estimation, although latency and clock period is not provided.

Joint software and hardware implementation is presented in [72]. Architecture is based on a control unit, neurons and shared Look-Up Table or LUT-based activation function. The implementation's control unit uses user-defined code to dynamically load weights and inputs, store neuron outputs and reset accumulators in neuron cells. The paper investigates two simple topologies with 1 and 4 neurons. The implementations are tested against time series prediction network, which uses a 2-4-1 structure. The provided solution's maximum performance is 0.66 μs and 0.44 μs for 1 and 4 neuron implementations respectively.

It is important to achieve reduced area and increased performance of the circuit, but this becomes increasingly difficult to carry out if low approximation error is required. In [73], the authors propose a hybrid approximation method of hyperbolic tangent activation function, which takes into account the linear nature of the hyperbolic tangent when the argument value is small. This approach is combined with a bit-level mapping of the function's non-linear region. Bit-level

mapping returns an average value of a sub-range of the region being approximated. Sub-ranges are split so that the approximation error is below a certain threshold.

The approach [73] is used in an optical character recognition system [74], where a FFNN is embedded into an FPGA. The authors use the aforementioned hybrid approximation method to implement a hyperbolic tangent activation function. The selected network's topology is 189-160-36, and the implementation's processing time is 4.36 μs.

A reconfigurable neural network architecture, composed of 20 neurons, is proposed in [75]. Architecture is divided into four parts: instructions unit, memory unit, layer unit and controller unit. Architecture adopts 8-bit precision. The approximation of the activation function is based on direct transformation from input to output. The network is tested using 4-8-3-3 and 1-5-1 topologies. The architecture is generic, i.e., it applies to different topologies without reconfiguration. The implementation uses a VEDIC multiplier instead of available on-chip Digital Signal Processing (DSP) blocks.

A valuable work exploring floating-point based implementation is carried out in [76]. Two approaches - resource-saving and parallel - have been developed. The exponent, used for hyperbolic tangent and sigmoid function, is calculated using Padé approximation. The approach is benchmarked with 5-16-12-16-5 topology and Xilinx ZEDBoard evaluation board with Zynq XC7020 chip. The authors illustrate that implementation is advantageous over high-performance software platforms due to its parallel execution.

In conclusion, the intrinsic programmable logic's parallel nature suggests its suitability for the implementation of FFNNs. Although different architectural approaches and design choices have been investigated, FFNN implementations face a major challenge of limited hardware resources. Furthermore, expansion of the NN topologies [77] and saturation of the manufacturing process improvement [78] suggest the persistence of the resource challenge. The summary of different topologies and performance metrics of previous FFNN design approaches is summed up in Table 2.1.

An interesting use case in terms of the potential application of our developed approach is presented in [8], where FFNNs enhance vehicle dynamics for a multi-motor electric vehicle. The authors train a predictive NN for the estimation of the future slip values of each wheel for a batch of possible torque-vectoring set points. These predictions determine the torque distribution that will reduce the unnecessary slip. Furthermore, the authors benchmark the trained topologies us-

Table 2.1. Summary of the NN topologies and the performance metrics from the related articles.

Paper	Data Type	Topology	Activation Function	Approximation Method	Latency (µs)	Throughput (Samples/s)
[71]	Fixed $< 16, 7 >$	2-2-1	Sigmoid/ Linear	Piece-wise linear	0.034	29,412,000 (theoretical)
[72]	Fixed $< 32, 7 >$	2-4-1	Sigmoid	LUT-based	0.44	2,272,700
[74]	Fixed	189-160-36	Hyperbolic tangent	Hybrid linear + bit-level mapping	4.36	229,360
[75]	Fixed $< 8, 3 >$	4-8-3-3	Hyperbolic tangent / linear	Direct mapping	1.16	862,070
[75]	Fixed $< 8, 3 >$	1-5-1	Hyperbolic tangent / linear	Direct mapping	0.683	1,463,100
[76]A	Single precision	5-16-12-16-5	Hyperbolic tangent / linear	Padé	33,100	30.2
[76]B	Single precision	5-16-12-16-5	Hyperbolic tangent / linear	Padé	24,700	40.5
[76]C	Single precision	5-16-12-16-5	Hyperbolic tangent / linear	Padé	5700	175.4
[76]D	Single precision	5-16-12-16-5	Hyperbolic tangent / linear	Padé	3500	285.7

ing parallel computing platforms. One of the trained topologies is used in this article to validate the developed approach in a virtual sensor use-case, which will be further described in Section 4.1.

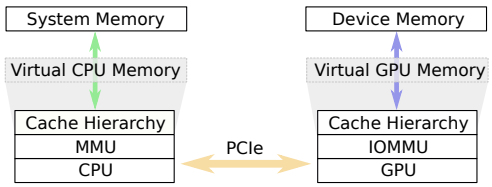
3. HETEROGENEOUS COMPUTING ARCHITECTURES

This section is devoted to distinct developments and advancements of computing architectures for HSoC-based systems and Linux in general. Most notably, an FPGA-master driven system architecture is proposed that also facilitates the final stereo-vision demonstrator. Also, a contribution in the field of real-time control loop systems is offered, where the designed AMP subsystem solution combines the functional benefits of the software stack available under the Linux operating system and real-time performance characteristics of bare-metal processing. Further, a set of software component management frameworks is proposed that facilitate the efficient implementation of the system architecture following a blackboard software development pattern, thus making frameworks suitable for design space exploration. Furthermore, the deployment of such demanding applications as autonomous driving and autonomous drones is briefly examined.

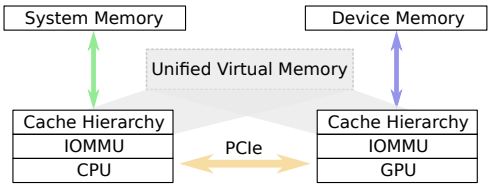
3.1 Heterogeneous Computing Based on Direct Memory Access

HSoC technology entices implementation of algorithms where every subtask executes on the most appropriate processing technology, i.e. processor is better suited for out-of-context tasks, such as decision making and control, and FPGAs excel at high-throughput number crunching and relatively local algorithms, for example, per-pixel operations, convolution and feature extraction. Furthermore, embedded systems follow a trend towards multi-peripheral processing, e.g. Nordic chips provide *Programmable Peripheral Interconnect* (PPI), which enables peripherals to interact autonomously with each other using tasks and events independent of the CPU while ensuring precise synchronization between peripherals for real-time applications [79].

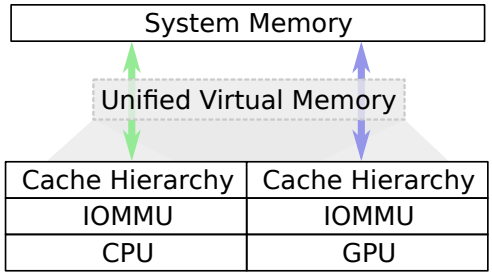
While such an approach activates processors less often and suggests lower energy consumption, the utilization of the co-processing methods can be more challenging due to the maintenance of cache-memory coherency and dealing with memory virtualization for application-level processors as described in Section 1.6. In pursuit of simplifying the programmability of SoCs, *Heterogeneous Systems Foundation* (HSF) has created *Heterogeneous System Architecture* (HSA) Memory Consistency Model where the host and the accelerator communicate via coherent shared memory [80].



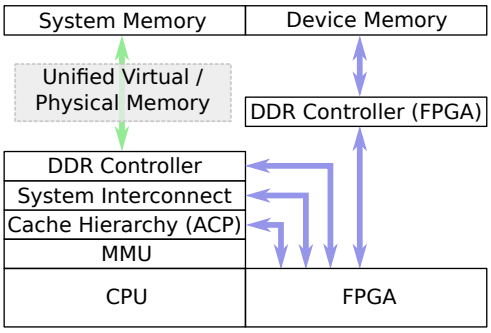
(a) Standard memory model with a discrete GPU attached to the PCIe bus. Model does not permit zero-copy between the GPU and CPU due to distinct physical memories [80].



(b) A unified virtual memory facilitates passing pointers over PCIe; nevertheless, zero-copy is still not possible [80].



(c) Unified main memory, where not only programming model is simplified, but there is no need to copy data between the GPU and CPU [80].



(d) Memory model and possibilities of modern HSoCs, where FPGA has access to separate memory, the shared system memory, internal interconnection logic and cached processor memory.

Figure 3.1. Examples of a standard CPU-GPU and HSoC memory coordination models.

Fig. 3.1a illustrates a standard memory coordination model that distinguishes between host and device memories; therefore, implying data copying from one master to another. More advanced models can have unified virtual memory where both masters can operate using the same addresses (Fig. 3.1b). Nonetheless, although that may simplify the programming model, the OS still manages data transfers between the master memories. A more advanced memory coordination model can be implemented when both masters share access to the system memory controller as shown in Fig. 3.1c, i.e. masters are implemented within a single chip. Opposed to the GPU technology, FPGA accelerators mostly are not processor-based; therefore, memory virtualization and context-switching mechanisms are not directly applicable. Furthermore, the high-granularity of FPGAs manifests itself into longer reconfiguration delays making it difficult to link it with the processor context-switching functionality.

Many modern HSoC devices provide a unique choice for memory coordination, as shown in Fig. 3.1d. FPGA may (1) implement its own DDR controller and utilize large amounts of isolated memory, (2) have direct access to the processing system's DDR controller enabling

performant yet non-cached data transfer, (3) have access to the system's interconnect, also enabling the utilization of processing system's *On-Chip Random Access Memory* (OCRAM) and (4) even access the cache-coherency ports that ensure memory access coherency on a hardware level. The aforementioned aspects provide a unique opportunity for the system designer while simultaneously creating a challenge for abstracting and controlling the accelerators from the userspace application.

In addressing the underlying challenges, FPGA-master based system architecture has been developed and benchmarked. The dual nature of the *Field Programmable System on Chip* (FPSoC) technology brings additional complexity into the system design. System architects are required to have considerable knowledge in both: hardware and software. Intel and Xilinx FPSoCs use an ARM-based processor that also incorporates *Single Input Multiple Output* (SIMD) NEON instruction set capable of floating-point operations. This encourages the development of complicated co-processing systems [81]. One of the most challenging issues considering the overall architecture of such designs is the creation of an efficient communications model between the processor and FPGA.

Micro-processors are capable of running Linux, which is one of the most popular operating systems for embedded devices, has considerably low requirements, supports a wide range of customization, provides multi-threading and has a variety of available open-source software accelerating the development [82]. Nonetheless, the usage of Linux implies additional design considerations due to the presence of virtual memory that might not have a meaningful impact on low-bandwidth systems. Nevertheless, when considering high-speed on-chip communications and co-processing architectures, the non-continuous nature of the data layout must be addressed.

Furthermore, the high complexity of HSoC technology has led to the creation of specialized integration tools and high-abstraction software for hardware description generation - *High-Level Synthesis* (HLS). HLS speeds up the development cycle but is not sufficient for architectures that exploit extreme parallelism and intermediate-level operations [83] and, in some use cases, HLS can significantly increase the number of needed FPGA resources [84].

To address the fundamental issue of HSoC on-chip communications, a DMA-based high-bandwidth communications architecture was developed for establishing the means of communicating between FPGA and software in an embedded Linux environment. The developed solution targets the Linux memory fragmentation issue and adopts the Linux kernel's *Contiguous*

Memory Allocator (CMA) feature. Furthermore, the data paths were benchmarked for Cyclone V SoC device as this data was still lacking in the scientific literature [85–87]. The developed modules and libraries have been made available to the public under MIT license[11].

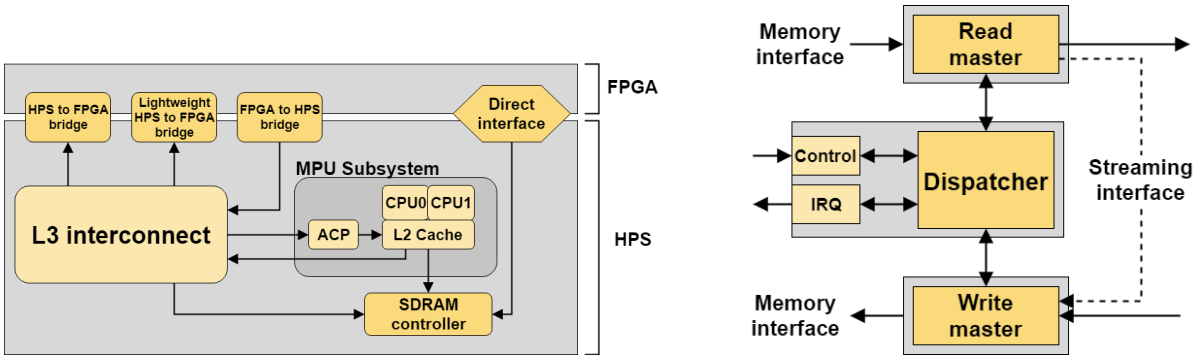
Cyclone V SoC [88] consists of two distinct portions– a single or dual-core ARM Cortex-A9 based *Hard Processing System* (HPS) and FPGA. The HPS architecture integrates a set of peripherals that reduce board size and increase the system's performance. Generally, the communication between HPS and FPGA in Cyclone V SoC devices can be accomplished using the following paths [88]:

- ***HPS-to-FPGA bridge*** – A high-performance interface from HPS to FPGA. Transactions are usually conducted by the processor or DMA controllers present in HPS. The bridge enables accessing the FPGA logic, peripherals and memory.
- ***HPS-to-FPGA Lightweight bridge*** – A low-performance interface to the FPGA fabric that usually is used by the processor to access control and status registers of the components implemented in FPGA.
- ***FPGA-to-HPS bridge*** – A high-performance interface from FPGA to HPS peripherals and memory. Cached memory transactions are supported by adopting ARMs *Accelerator Coherency Port* (ACP).
- ***FPGA-to-HPS SDRAM interface*** – A high-performance interface from FPGA to the HPS *Synchronous Dynamic Random-Access Memory* (SDRAM) controller. FPGA master has access to the processor's *Random-Access Memory* (RAM). Data residing in the processor's cache will result in errors, an issue that must be addressed by the software.

An important aspect of the particular system's structure is the Level-3 (L3) interconnect, which is a central switch that routes data between memory, FPGA fabric, processor and peripherals [88]. Fig. 3.2a shows the relevant aspects of the Cyclone V SoC's architecture.

In the particular setup, Altera's *Modular Scatter-Gather Direct Memory Access* (MSGDMA) controller fulfils the role of the FPGA master; it consists of reading and writing master subcomponents and a dispatcher core [89]. Fig. 3.2b shows the simplified structure of the controller. Read and write masters communicate with memory using Avalon memory-mapped interface. Dispatcher manages transactions and is configurable by software via HPS-to-FPGA Lightweight

[11] http://git.edi.lv/rihards.novickis/FPSoC_Linux_drivers



(a) Relevant Cyclone V interconnect connections. **(b)** Modular SGDMA controller IP.

Figure 3.2. Internal blocks related to the DMA master-based architecture.

bridge. The configuration also enables the generation of interrupt requests upon finishing the transactions. The software utilizes this feature to measure data transfer timings. During synthesis, the MSGDMA core also can be configured to one of three transfer modes:

- memory mapped to memory mapped transfers;
- memory mapped to stream transfers;
- stream to memory mapped transfers.

The specification of the utilized Avalon memory-mapped interface was created by Altera company to simplify and accelerate the FPGA design process [90]. Moreover, on-chip communications in ARM systems are defined by the AMBA specification [91]. These particular communication protocols are bridged together using high-bandwidth structures – Qsys interconnects [92].

As described in section 1.6 Linux operating system, utilization of virtual memory mechanism and process management by software resolves into memory fragmentation, and over time larger chunks of contiguous memory become unavailable. While virtual memory enables multiple concurrent processes, memory discontinuity is an issue for DMA since peripherals operate in physical address space [93]. Contiguous memory acquisition involves standard driver system calls, such as *kmalloc()* (with GFP_DMA flag) or *dma_alloc_coherent()*. Usually, this approach is limited to a few megabytes, but high order requests are prone to fail even when the requested buffer is less than 128 KB due to memory fragmentation [93].

An alternative is to allocate a buffer in smaller pieces and use scatter/gather memory access. Although the MSGDMA controller acts as an FPGA master, another approach is encouraged. Usually, large DMA buffers are reserved at system boot time, and if the device is not using

this memory region, it stays unused [94], which may lead to insufficient resource utilization. Memory continuity is achieved by applying CMA feature [93] that enables dynamic allocation of large buffers (>50MB) suitable for benchmarking and efficient memory utilization in real-life applications. For this purpose, a custom Linux driver has been developed that allocates contiguous memory and provides mechanisms for transferring control to the user-space. Another custom driver ensures the control of the MSGDMA dispatcher; it also supports multiple user-space applications and event handling.

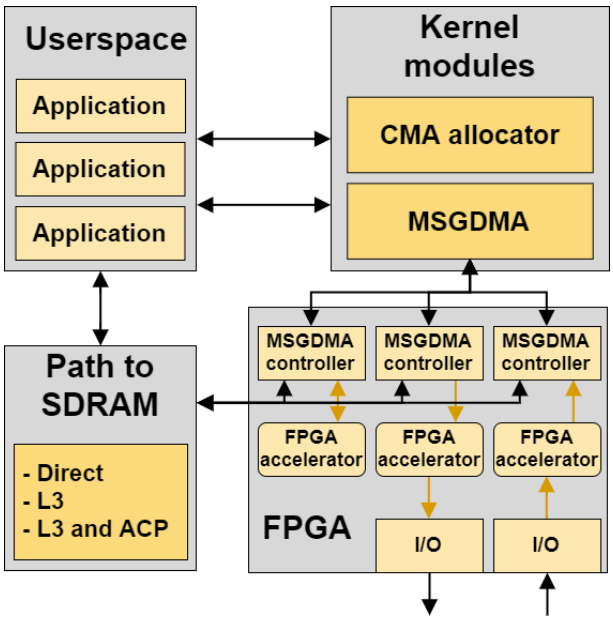


Figure 3.3. Conceptual design of FPGA Master-based architecture.

All the above considerations constituted the development of the modular FPGA Master-based architecture shown in Fig. 3.3, where user applications are able to:

- dynamically allocate contiguous memory,
- reserve MSGDMA resources,
- initiate DMA transactions,
- are suspended by the driver when the transactions start and awoken when transfers have ended (interrupt support).

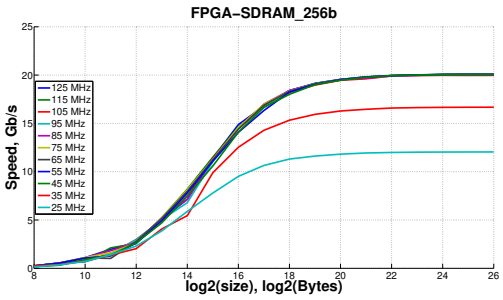
Finally, the throughput measurement procedure utilized the VEEK-MT-C5SoC development board for all FPGA master communication bridges by adjusting the interface bus width, transaction size and FPGA clocking frequency. The timing measurements utilized the internal *Snoop Control Unit* (SCU) timer via call to *clock_gettime()* function. Measurements incorporate the

execution of the MSGDMA controller and software that corresponds to usage in real-life applications. All tests were run ten times, and a full set of measurement results is provided in Appendix 3. Table 3.1 shows the maximum (saturated) throughput for each of the data path configurations and Fig. 3.4 shows the nature of throughput measurements at the widest configurable bus widths and varying FPGA clock frequencies. The power consumption was measured in different setup where tests were run continuously for 300 seconds, and the measurements where taken using the onboard power monitor with a total unadjusted error of $\pm 1.0\%$, but no distinguishable results were found. Power consumption in the idle state was 8.11 W, but for any of the data path scenarios, it was always 8.18 W within the measurement error range.

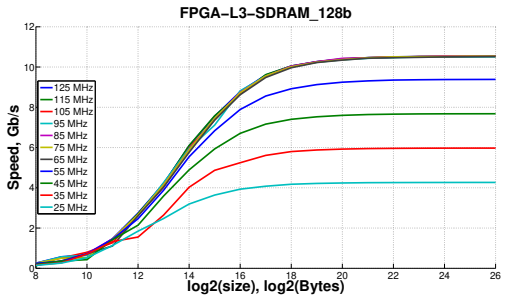
Table 3.1. Throughput of simultaneous read/write transactions for all communication interface configurations.

Data path	**Bus width**	**Maximum throughput**	**Saturation frequency**
FPGA-L3-SDRAM	32 bits	5.05 Gbps	120 MHz
	64 bits	10.10 Gbps	120 MHz
	128 bits	10.52 Gbps	65 MHz
FPGA-L3-ACP-SDRAM	32 bits	6.90 Gbps	-
	64 bits	8.64 Gbps	120 MHz
	128 bits	11.26 Gbps	90 MHz
FPGA-SDRAM	32 bits	7.52 Gbps	-
	64 bits	14.64 Gbps	-
	128 bits	17.68 Gbps	80 MHz
	256 bits	20.08 Gbps	45 MHz

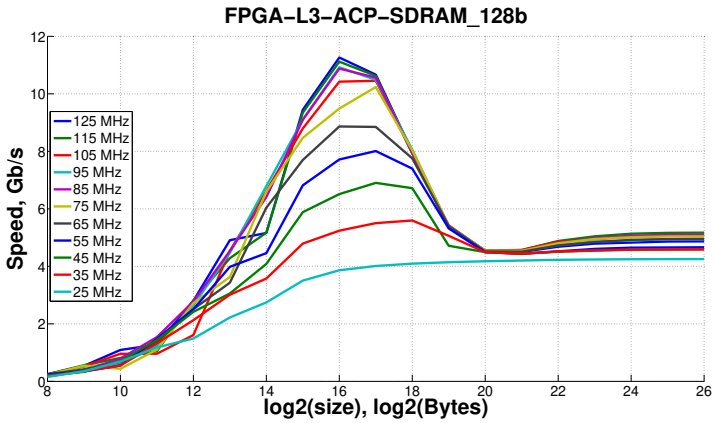
In summary, the examined technology (Intel Cyclone V SoC) can sustain on-chip communications bandwidth of 20.08 Gbps, which is suitable for real-time image co-processing. Fig. 3.4c shows a curious result where communication bandwidth hits a peak and then drops to saturation. This characteristic is explained by the nature of the particular communication scenario, as, in this case, the transactions are first hitting the ACP, which then forwards the requests to the L2 cache. If the request ”hits” the cached data, it is returned immediately, but if encountered with a *cache miss*, the transaction is forwarded further to the *Double Data Rate* (DDR) controller. Because the L2 cache size for the Intel Cyclone V SoC is 512 kB and tests utilize simultaneous writes and reads (the cache simultaneously holds data from both - read and write memory regions), the throughput peaks at the half size of the cache.



(a) FPGA-SDRAM interface at 256 bit bus configuration.



(b) FPGA-L3-SDRAM interface at 128 bit bus configuration.



(c) FPGA-L3-ACP-SDRAM interface at 128 bit bus configuration.

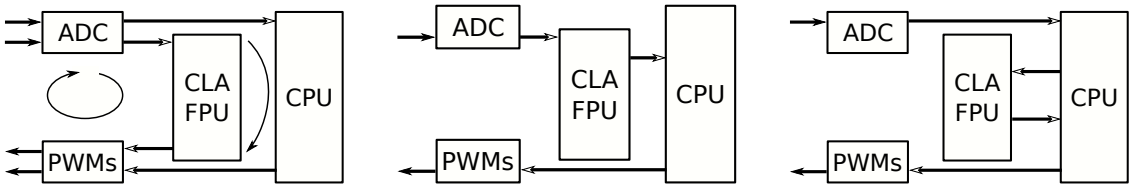
Figure 3.4. Throughput measurements for different FPGA master communication paths.

3.2 Approach of Asynchronous Multi-Processing

A considerable contribution of this thesis is the novel employment of heterogeneous architectures for real-time control loop systems by using an approach based on the *Asynchronous Multi-Processing* (AMP) subsystem. While FPGA technology is a well-suited medium for real-time control due to its determinism, acceleration of such computational tasks where the processing involves leaping across memory can be problematic or even unsuitable, e.g. server applications, recursive algorithms, complex decision making. These considerations were addressed within the H2020 ECSEL I-MECH project by designing a real-time application subsystem that combines application-level functionality while not compromising the determinism of the processing hardware.

I-MECH project was devoted to modern motion control systems, which usually are designed in a distributed manner. There is a central controller that manages motion trajectory planning and higher-level *Multiple-Input, Multiple-Output* (MIMO) control loops. Actuators, sensors and

motor drivers are connected to the central controller with high-speed real-time data communication interface, .e.g. EherCat [95]. Simplified examples of different embedded control loop configurations is shown in Fig. 3.5. Although EtherCAT is capable of running communication cycles under 50μs [96], the conventional platforms are usually based on multicore x86-64 CPUs that provide large computing performance but are not applicable for non-standard I/O technology required for SoA applications [97]. By acknowledging the requirements of latency, reliability, supporting local feedback loops and application-level customization, the I-MECH project came up with a concept - the modular signal processing unit for motion control applications based on HSoC technology [97]. The developed AMP subsystem is a part of this modular processing unit.



(a) Running parallel control loops. **(b)** Preprocessing input data. **(c)** Co-processing with the main CPU.

Figure 3.5. Texas instrument embedded control loop's configuration examples using their *Control Law Accelerator Floating Point Unit* (CLA-FPU) [98]. The *Analog-to-Digital Converter* (ADC) measures feedback signals and triggers control-loops execution, the *Pulse Width Modulation* (PWM) represents the outputs of the system.

The asymmetric multiprocessing (AMP) refers to a multiprocessor computer system where different processing units are treated unequally. The processing cores may have different architectures (heterogeneous or homogeneous multicore) and may run various operating systems. The AMP is likely to be used when different CPU architectures are optimal for specific activities. The AMP approach also applies for mixed-criticality use-cases, when, for example, the critical code runs on a separate core, which also simplifies the certification of the system as only the critical code needs to be certified. Commonly, a software layer - hypervisor - is added that administers multiple operating systems, including managing accessible hardware and even prioritizing bus access, thus effectively configuring the real-time characteristics of the different cores. The AMP cores may also have some inter-core communication facility, e.g. through shared memory and inter-core interrupts [99]. While the hypervisor helps set up AMP system, it negatively affects the performance. For example, Byoungwook and Min [100] demonstrated

that on the NVIDIA Jetson TK-1 platform, the hypervisor introduces substantial delays in task switching (from 3.5μs to 4 ms) and interrupts (from 3μs to 150μs).

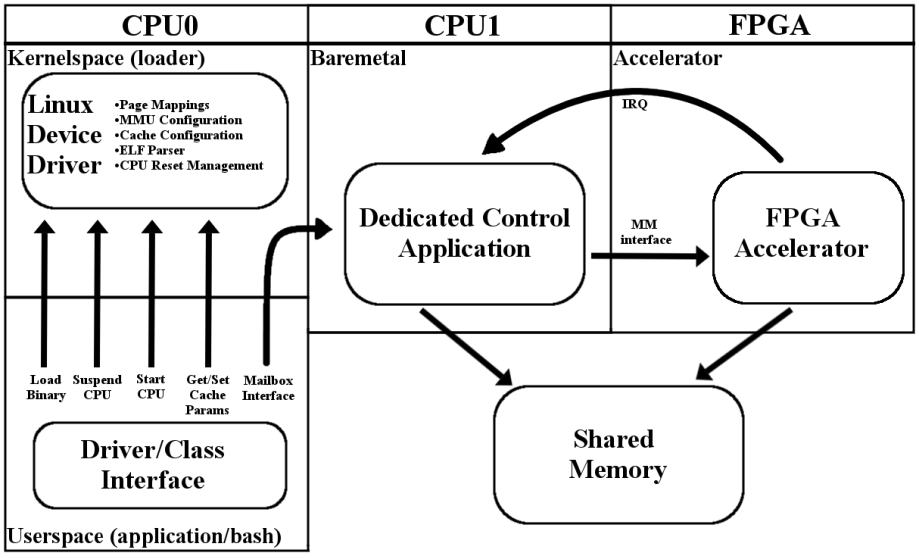


Figure 3.6. Real-time application subsystem.

The developed overall AMP solution aspires to provide the best of the two worlds: real-time processing capabilities provided by the AMP core and FPGA and functionality of the Linux software stack. Fig. 3.6 illustrates the overall concept of the developed AMP subsystem; in this dual-core processing system, CPU0 executes the operating system (Linux) while the CPU1 (AMP subsystem) cooperates with FPGA and collaboratively executes RT tasks. The interface to the AMP subsystem is implemented as a Linux driver (illustrated in Fig. 3.7), and it ensures the following functionalities:

- controlling the AMP core (putting the core on reset or pausing its execution),
- loading baremetal *Executable Linked Format* (ELF) applications for execution on the AMP core (including the configuration of the virtual memory),
- providing diagnostics for the execution of the application,
- providing means for on-the-fly configuration of the application,
- communicating with the AMP core (*stdin*, *stdout*, *stderr*).

In an ARM-based processor, some aspects of the configuration, e.g. enabling interrupts, bringing up caches and MMU, can be set only by the CPU core itself, i.e. via co-processor configuration interface [101]. AMP subsystem setup can be comprehended by examining the booting process of a particular SoC. Although, SoCs have distinct internal structures, the mech-

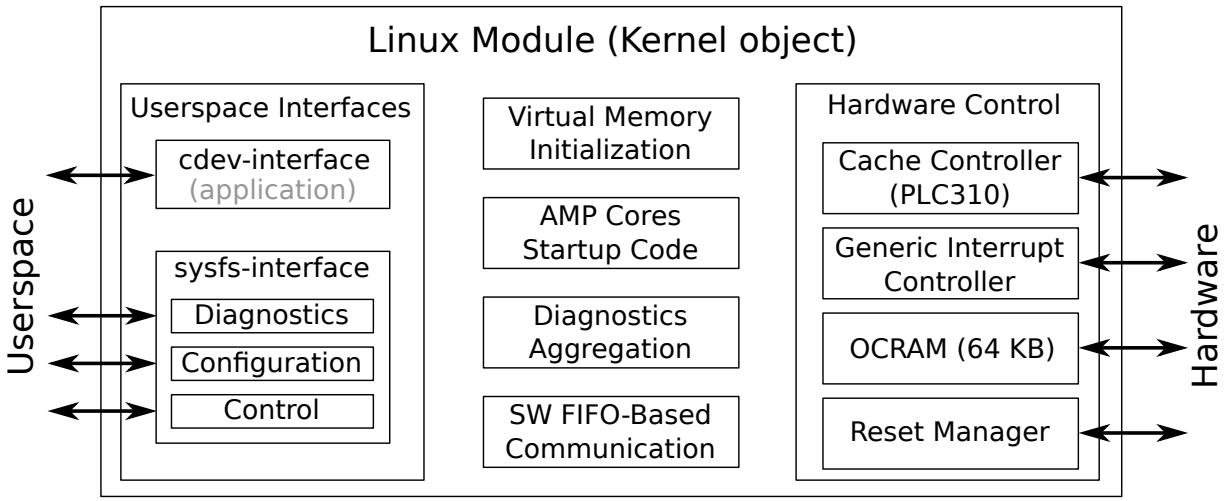


Figure 3.7. Composition of the developed Linux driver.

anisms have commonalities, therefore lets use Intel's Cyclone V dual-core HSoC [88], the relevant parts of the architecture are illustrated in Fig. 3.8.

In a conventional setup, when the system returns from the reset state, the first core fetches instructions from the *0x00000000* physical address. SCU, which is ensures cache coherency between the processor cores, passes this data request to the Level 2 (L2) cache controller. L2 controller utilizes a "filtering" feature, which steers requests either to Level-3 (L3) interconnect or DDR memory controller. The default (reset) configuration propagates the transactions below 1MB to the L3 interconnect, which furthers it to the *On-Chip Read Only Memory* (OCROM), which holds the startup code written by Altera. This code is responsible for copying the next program stage into OCRAM and passing it control. Eventually, startup code configures the DDR controller and re-configures the L2 controller's filtering feature to propagate low address requests to the DDR controller.

Therefore, bringing up the core during the runtime requires the control of the *0x00000000* physical address, which during the execution of the Linux targets the DDR memory. By planting custom code at this address, it is possible to kick off the bootup of the AMP core and the respective RT application. By default, the caching mechanisms are disabled; therefore, it is also necessary to flush L1 and L2 caches after deploying the core's bootup code. It may be beneficial to place the RT application in the OCRAM as it is faster than the DDR memory; nevertheless, it is limited to 64 KB, and the developed solution utilizes it for inter-core communications. Unfortunately, only the processing core can invalidate and enable its L1 cache and MMU; therefore, this functionality must be executed as a part of the RT application or its startup code.

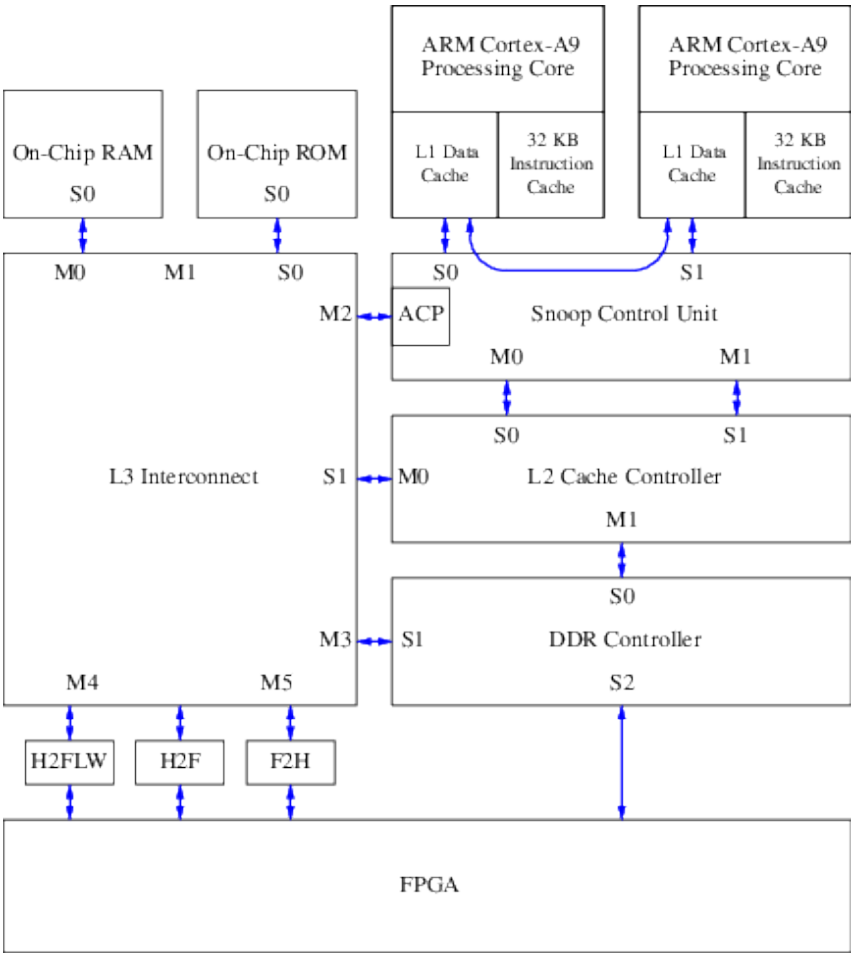
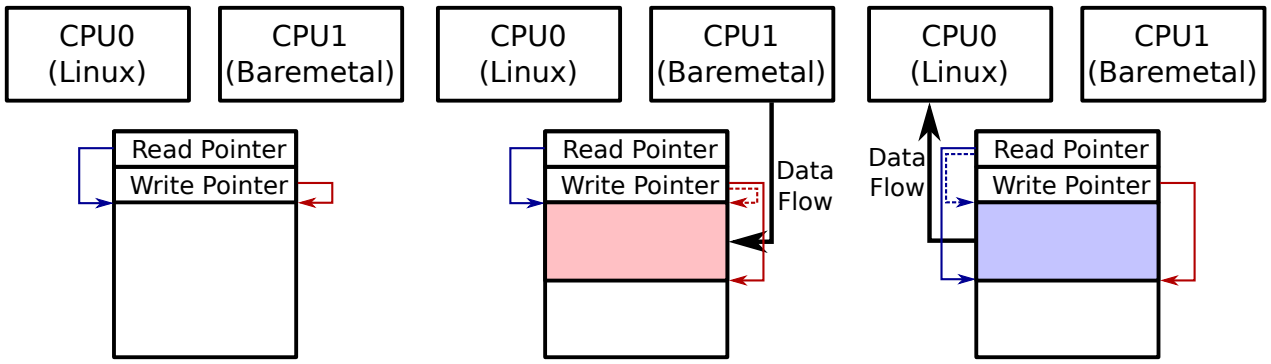


Figure 3.8. High-Level Structure of Intel Cyclone V Field Programmable SoC.

Consequently, after the AMP core's setup, it is necessary to incorporate some inter-core communications mechanisms. The straightforward approach is to use the shared memory mechanism, where the processor cores communicate via some predefined protocol. By itself, such an approach can be ineffective, as it implies periodic memory access or even polling, which also harms the real-time characteristics of the system bus. Conventional processing systems and interrupt mechanisms incorporate means of sending *Inter-Processor-Interrupt* (IPI) as this would enable asynchronous communication. When developing the AMP subsystem, the Linux IPI configuration is separate from the rest of the interrupt mechanism and, therefore, Linux kernel code required some minor modifications. The designed inter-process communication mechanism utilizes software-based *First-In-First-Out* (FIFO) implementation and the asynchronous IPIs (the working principle is illustrated in Fig. 3.9). Simultaneously several such FIFOs are implemented, establishing the conventional *stdin*, *stdout* and *stderr* data streams for the RT application.



(1) Communications FIFO is initialized. Read/Write pointers point to the start of the shared buffer (physical address).

(2) Communications master core wants to communicate. It fills the buffer with data, updates the write pointer, flushes caches to the point of common coherency and initiates IPI to the slave core.

(3) Slave core has received communications IPI. It reads and interprets data, updates read pointer and flush caches to the point of common coherency.

Figure 3.9. Inter-processor communications mechanism.

Another feature of the designed AMP subsystem is the runtime capability to change the configuration. The method utilizes the implemented FIFO communication mechanism and custom segments in the ELF file. During the AMP core setup, the driver parses the custom segments and configuration options are made available to the user using the sysfs interface. From the user's perspective, the application code can utilize *read-only* and *read-write* configuration using C-macro functions, e.g. to define read-only VERSION string and DELAY unsigned integer variables, one would use:

```
CONFIG_RO_CSTR(VERSION, "0.1.1");
CONFIG_RW_UINT(DELAY, 125);
```

A similar concept is utilized in the designed Linux software component management approach, described in more detail in Section 3.3 Approach to Management of Software Components.

As for the user perspective, the driver provides the standard *cdev* (character device) and *sysfs* (pseudo-filesystem) interfaces. The *cdev* interface is used to upload the new ELF application onto the AMP subsystem. During this process, the driver (1) puts AMP core into the reset state, (2) allocates memory for the new executable, (3) parses ELF file, (4) copies the application to memory while simultaneously initializing the MMU page table, (5) updates entry points and MMU translation page address in the startup code, (6) parses and initializes configuration API, (7) flushes caches and (8) unsets core's reset. While it is possible not to use the MMU, the built-in

mapping provides an additional layer of security and isolation, including controlling the access to hardware resources and isolating AMP execution from the operating system. This interface is also utilized for providing the *stdin*, *stderr* and *stdout* streams of the baremetal application.

The *sysfs* interface provides access to the internal kernel data structures, i.e. from the user's perspective, the communication is through a set of directories and files that resemble the internal kernel object (driver) structures defined internally within the kernel [102]. This interface provides access to the configuration of the application, control of the AMP core and control loop's diagnostics (minimum, maximum and average control loop's lengths). Opposing the standard approach for exchanging custom data structures across userspace and kernelspace, i.e. *ioctl*, the user can select any programming language for interfacing with the system, e.g. bash, python, C.

3.3 Approach to Management of Software Components

As a consequence of building and investigating Linux-based multi-component (process) systems is the development of a novel component-based software architecture concept, which already has been successfully applied for autonomous vehicles and drone systems in H2020 ECSEL PRYSTINE and *Framework of Key Enabling Technologies for Safe and Autonomous Drones* (COMP4DRONES) projects. While developing a complex and evolving system, it may be necessary to have an architecture that facilitates effortless software component management and provides convenient means for their inter-communication. This approach resembles a "blackboard" pattern from the software development theory [103], which anticipates non-determinism of the software-component final composition. Furthermore, the processing capability requirement may evolve; thus, the system architecture should also be scalable.

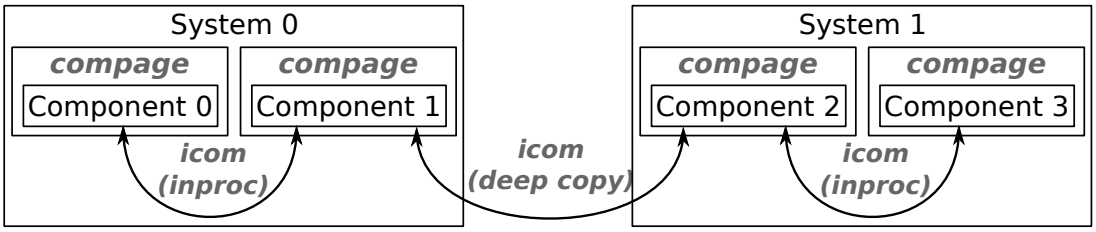


Figure 3.10. An approach to the management of software components.

Fig. 3.10 represents the principle of the developed system's component management and inter-communication solution. The developed approach utilizes two modular frameworks:

- ***compage*** (component management framework) - ensures component management within a single system, including the initialization and execution of the components in separate threads or processes. The framework also provides the means of generating and applying component configuration using *.ini* files. The developed framework is made available to the public at: https://gitlab.com/rihards.novickis/compage.
- ***icom*** (component communication framework) - ensures coherent communication between different software components by providing various underlying communication mechanisms, thus enabling zero-data copy whenever possible. Zero-data copy excels when the processing pipeline involves a large quantity of data, which, if not shared efficiently, can result in a substantial performance drop due to the saturation of the on-chip communications backbone (interconnect). Essentially, if software components execute on a single system, they can share data references and avoid data duplication, e.g. utilizing the double buffer technique [104]. The developed framework is made available to the public at: https://gitlab.com/rihards.novickis/icom.

The core reasons for the development of such frameworks are the shortcomings of the SoA solution - *Robot Operating System* (ROS) [105]. Although ROS has become a widely adopted standard in robotics research, it is not suitable for embedded control systems [106] mainly due to its real-time characteristics and size. Notably, ROS2 overcomes some limitations of its predecessor [107], and there are also initiatives for bringing ROS to embedded systems, e.g. H2020 project - Micro-ROS: Platform for seamless integration of resource-constrained devices in the robot ecosystem.

3.3.1 Software component management framework - *compage*

The component management framework was set to meet the following requirements:

- the framework shall have a small footprint, marking it suitable also for embedded systems;
- the framework shall provide means of configuring and replicating software components;
- the framework shall be integrable in both *C* and *C++* workflows;
- the framework shall be manageable, i.e. adding new components shall require minimum effort.

compage framework addresses these requirements by creating a custom segment in the ELF executable, with a structure as shown in Fig. 3.11. The software component essentially is a

function (entry point) with a predetermined and configurable data structure as its argument. The framework permits denoting such an entry point, its default associated data structure and marking specific parameters for configuration. The framework can express component composition using an *ini* configuration file that can further be generated (the default configuration), modified to change parameters or duplicate software components and used as input to execute the described system.

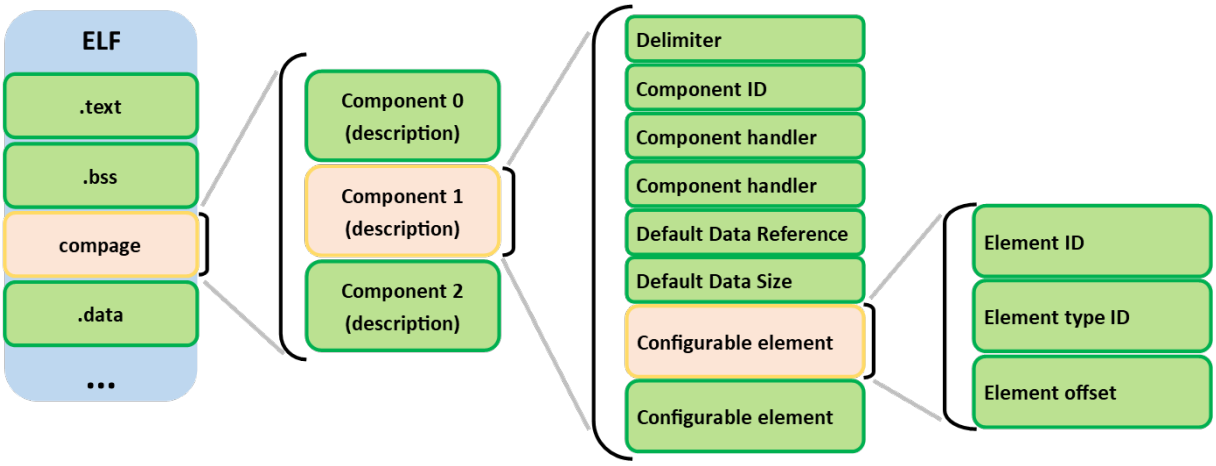


Figure 3.11. Layout of a *compage* segment in the executable.

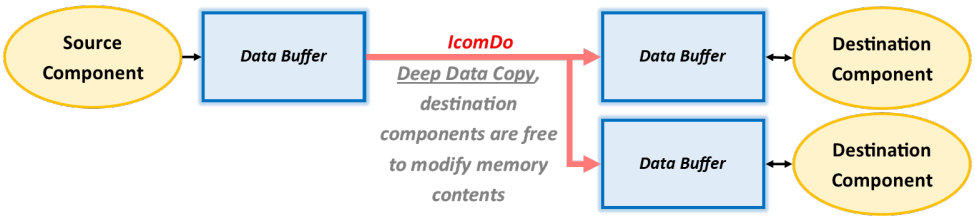
An example of framework's instantiation is shown in Appendix 6, Appendix 7 illustrates description example of a simple *compage* component and generated / modified configuration file is shown in Appendix 8. The *compage* development flow can be summarized as follows:

1. The framework is integrated into the main program flow by utilizing its API.
2. New components are developed and registered with the framework. Notably, each component's code may be located in a single file.
3. The framework generates the default configuration file.
4. User modifies configuration file to update component parameters and instantiate multiple copies of the same component.
5. The configuration file is supplied to the executable to run a component-based application.

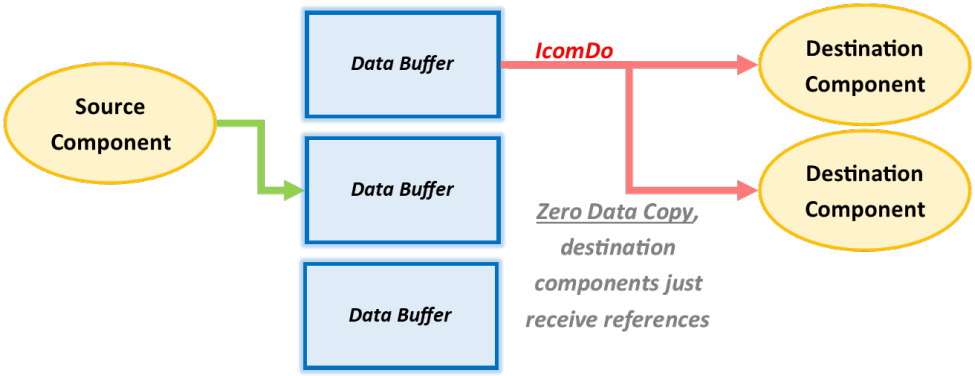
3.3.2 Software component communication framework - *icom*

The blackboard paradigm also requires a universal communication interface; nonetheless, there must be some context awareness to enable seamless switching between zero and deep copy approaches (Illustrated in Fig. 3.12). The developed *icom* framework originally was based on

Zero Message Queue (ZMQ) [108] framework, but extends to other IPC mechanisms, e.g. FIFO, message queues, sockets, shared memory, etc. The framework supports the standard PUSH-PULL, PUBLISH-SUBSCRIBE and REQUEST-REPLY communication paradigms. Compared to ROS-based solutions, this framework is suitable for deployment in complicated perception-based control systems, .e.g. high-level perception and control for autonomous drones (approbated in COMP4DRONES project) and vehicles (approbated in PRYSTINE project).



(a) Communication based on deep data copy.



(b) Communication based on zero data copy.

Figure 3.12. Component communication mechanism base.

An abstract example for using *icom* framework is provided in Appendix 9.

4. ADAPTATION AND IMPLEMENTATION OF COMPUTER VISION ALGORITHMS

This section illustrates the application of HSoC technologies in stereo-vision algorithm implementation. Currently, ML algorithms dominate many image processing applications; therefore, the initial discussion focuses on fully connected FFNN implementation in programmable logic. While the designed solution applies to torque vectoring algorithms of multi-motor electric vehicles, the gained insights illustrate the limitations of NN algorithm pipelining. The section further explores the developed heterogeneous system architecture for executing stereo image processing and discusses the details of implemented algorithms, i.e. image deinterleaves logic, interpolation of Bayer's pattern, spatial image transformation logic (for barrel distortion correction and rectification), feature extraction and disparity calculation. A core contribution - spatial image transformation - is discussed in more detail as it utilizes a novel approach to ensuring parallel data access that is generalized for N-dimensions and enables SoA interpolation while conserving OCRAM resources.

4.1 An Approach of Feed-Forward Neural Network Throughput-Optimized Implementation in FPGA

4.1.1 Design considerations

One contribution of the 3Ccar project focused on algorithms for enhancing vehicle dynamics on multi-motor electric vehicles, which benefit from better controllability offered by such powertrains. A part of this work is the development of a novel solution to the FFNN implementation challenge. The proposed approach proposes to revise the implementation challenge viewing it in terms of elementary structures and utilizing pipelining paradigm. The FFNN is split into elementary layers, where each layer can be characterized by its resource, e.g. adder, multiplier, activation function. These layers reserve different numbers of resources to achieve even distribution of latencies and attain an optimally pipelined implementation, where every layer's latency is less or equal to the time necessary for the network to accept new input. An accompanying tool, developed for the solution, converts the given network's topology into C++ code that further feds into an HLS tool. The generated code already incorporates necessary directives to ensure the envisioned solution with the requested pipelining iteration interval.

As a consequence of the NN parallel nature, a fully pipelined implementation can require more resources than is available. Furthermore, the pipeline's throughput depends on its slowest function, thus provided with a limited communications bandwidth, a fully pipelined implementation can be wasteful. As described in Section 2.5, FFNNs abstract a combination of neurons, and if we denote delay with τ and take into account data dependency, as illustrated in Fig. 2.9, neuron's delay can be characterized with Eq. 4.1:

$$\tau_{total} = \tau_{mul} + \tau_{add} + \tau_{act}, \tag{4.1}$$

where τ_{mul}, τ_{add} and τ_{act} are the respective delays of the multiplication, addition and activation operations.

By studying NN structure and assuming every operation is characterized by some operation-specific constant delay τ_c, we can derive the corresponding delay-resource relationships. For example, for the multiplication layer, all operations can be done in parallel; therefore, the delay model of such layer is:

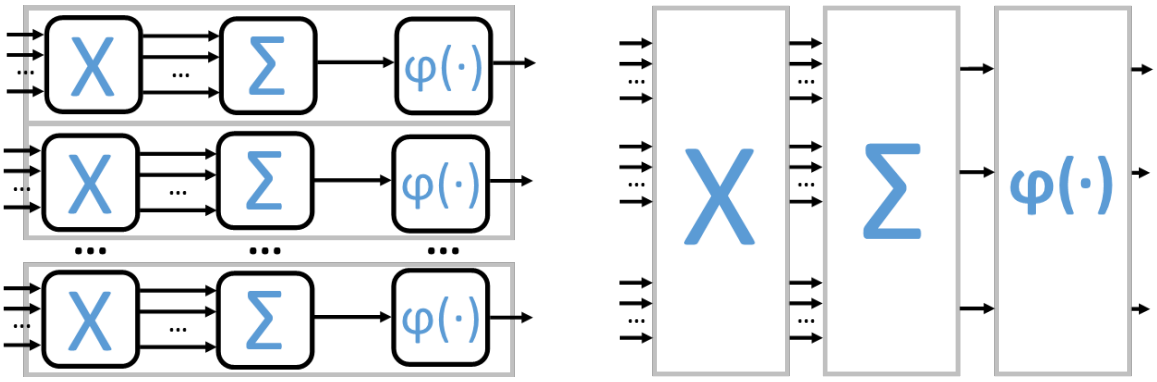
$$\tau_{mul} = \left\lceil \frac{N_{in}}{N_{mul}} \right\rceil \times \tau_c. \tag{4.2}$$

As for the summation phase, the computational process involves additional dependencies because all inputs are added together (bias is treated as one of the inputs). Depending on the clock frequency, chosen data type and routing, sometimes the adders may be cascaded, but such an approach requires additional information regarding the clock frequencies, adder widths and the technological characterization of the FPGA chip and, therefore, such level of detail is not analyzed. In the simple case, the minimum delay of the addition operations can be achieved by adopting a tree-like adder structure, and it is:

$$\tau_{add} = \lceil log_2(N_{in}) \rceil \times \tau_c. \tag{4.3}$$

In general, if the adder count is limited, the adder layer's delay can be determined with Algorithm 1.

Although neuron is an intuitive abstraction, it subdivides architecture as shown in Fig. 4.1a. This approach omits resource sharing between parallel neurons. Therefore a different abstraction is proposed, which is shown in Fig. 4.1b. In this approach, neural network structure is separated into elementary layers, where each layer is characterized by a specific resource - adder, multiplier or activation function.



(a) Non-optimized delay model for a single layer.

(b) Optimized delay model with resource sharing for a single layer.

Figure 4.1. Different delay models. a) Delays are analyzed in terms of a neurons in layers b) Joint delay analysis for ”primitive” multiplication, addition and activation layers.

By considering the main bottlenecks, e.g. limited bandwidth or resource availability, it is possible to optimize the network by allocating an appropriate amount of resources for every layer. One of the most common limitations in FFNNs is the activation function that often is difficult to pipeline. Furthermore, applying the elementary layer abstraction increases the complexity in adder scheduling, as different resource sharing arrangements can lead to different delays. This is illustrated in Fig. 4.2.

Algorithm 2 provides a procedure for delay calculation that considers the unused resources in one stage and carries them to the next one. Variables are designated as follows: N_{lin} - elementary layer's input count, N_{add} - adder count, N_{uadd} - adders used in the current stage, N_{cadd} - adders carried to the next stage, τ - delay cycles.

Another important consideration for the ANN implementation is the choice of data type, as floating-point data types contribute high precision and range but are costly to implement and pipeline. Furthermore, recent studies [109, 110] show that fixed-point data types can be used with a relatively small precision loss, especially if the ANN training procedure is aware of the

Algorithm 1 Addition delay for single neuron

1: $N = N_{in}$

2: $\tau = 0$

3: WHILE $N > 1$

4: $\quad N = N - min(N/2, N_{add})$

5: $\quad \tau + +$

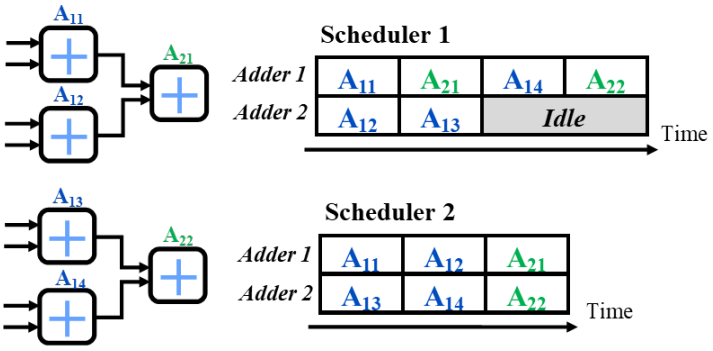


Figure 4.2. Illustration of different resource sharing policies where two adders are being shared between two neurons. The first scheduler prioritizes a neuron, which results in additional delay due to dependencies. The second approach prioritizes all neurons equally, leading to a more efficient resource sharing policy.

Algorithm 2 Delay of elementary addition layer

1: $N = N_{lin}$
2: $\tau = 0$
3: $N_{uadd} = N_{add}$
4: WHILE $N > 1$
5: $N_{cadd} = N - N_{uadd}$
6: $N_{uadd} = min(N/2, N_{add} + N_{cadd})$
7: $N = N - N_{uadd}$
8: $\tau + +$

fixed-point data type.

Digital ANN implementations also involve the coefficient storage trade-off. Coefficients can reside in registers, which ensures parallel access to all coefficients, or they may be stored in the on-chip memory, thus limiting their accessibility. The choice depends on the pipeline's performance requirements and also the size of the NN.

4.1.2 Design, implementation and results

A tool has been developed to automate FFNN implementation that takes the FFNN topology as an input and generates C++ code for the Xilinx HLS. The tool uses the following methodology: first, it splits network topology into "elementary" layers, where each layer is associated with a specific resource, i.e. adder, multiplier, activation function, etc. Resource sharing is accomplished with multiplexing logic as shown in Fig. 4.3, the implementation of this logic is left to the HLS. Further, the network delay model is derived. This information supports the latency

calculations for every elementary layer and enables making choices regarding the pipeline design. The tool reduces layer delay by increasing the number of its allocated resources; in an optimal pipeline, all layers should have the same delay - a principle that guides calculations behind throughput optimization.

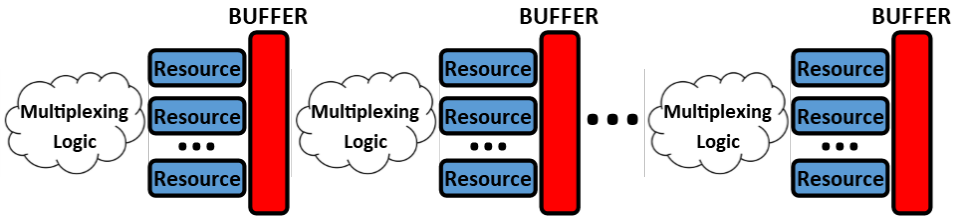


Figure 4.3. Proposed resource implementation scheme.

One of the main parameters of the tool, steering the process of the code generation, is the maximum delay acceptable for pipeline's stages, which is given in clock cycles. The parameter ensures allocating "just enough" resources to comply with the given constraint - the pipeline's initiation interval. Fig. 4.4 shows an example of such a relationship, where the same FFNN topology is fed into the developed tool while targeting the range of initiation intervals. Furthermore, the developed tool supports different normalization, addition, multiplication and activation layers. Activation function can be calculated either analytically by utilizing Xilinx and LOGICore IPs or by implementing a simple *Look-Up-Table* (LUT).

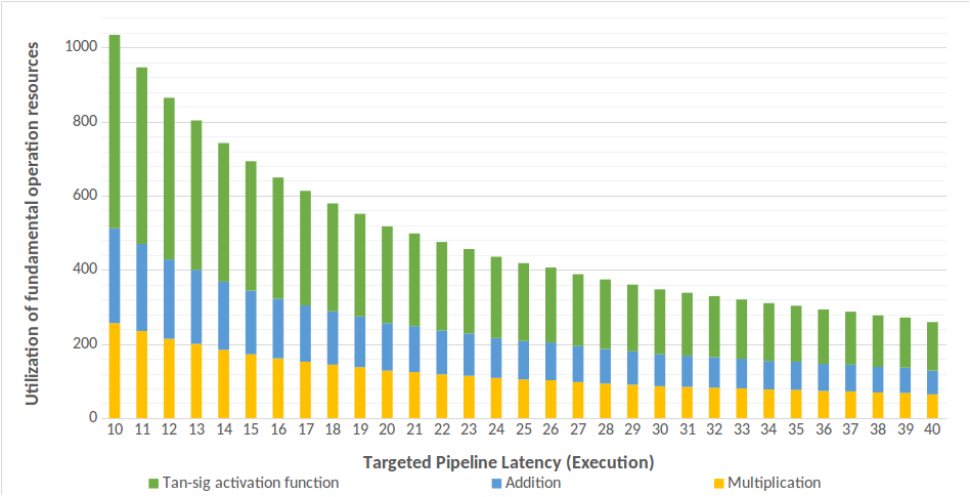


Figure 4.4. "Elementary layer" resource dependence on the targeted pipelining latency for a 17-40-30-20-4 FFNN topology. The normalization layers are not displayed, because their usage varies minimally (1-2 resources).

The generated code consists of header, source and data files and every elementary layer is

abstracted as a function. Source code incorporates directives, which guide hardware implementation. The developed tool is open-source and made available on-line[12]. Generated FFNN IP cores are tested using two hardware interfaces: *Memory-Mapped* (MM) interface for latency estimation and *Streaming* (ST) interface for latency and throughput estimation. MM interface implies active control from the side of *Micro Processing Unit* (MPU), but ST interface is set up by utilizing LogiCORE DMA IP cores [111] and AXI high performance interface [112].

In summary, the developed methodology implements FFNN by splitting the neural network into elementary layers, generating a high-level description of the topology and adopting HLS tools for the FFNN IP core generation. The developed workflow is designed for optimal throughput and, although it is not suitable for fully pipelined image processing, the solution applies for virtual sensor implementation as it can provide a relatively high sampling rate.

All tests were performed on the Xilinx Zynq ZC702 SoC evaluation board using a bare metal stack with FPGA logic clocked at 100 MHz frequency. Timing measurements are made by using the Cortex-A9 SCU timer. MM-based accelerator variant of the core is benchmarked by running IP core 100 times and measuring its time of operation. Timer measurements are made before launching IP and after receiving the interrupt signal indicating the end of the core's operation. Notably, there are sub-microsecond measurement errors brought in by the internal interconnect structure. The ST interface was tested 100 times with 10,000 continuous neural network data points for estimating throughput and 100 times with just a single data point for estimating latency. The performed tests and detailed comparison with other solutions are provided in Tables 4.1 and 4.2.

Results in Table 4.1 illustrate that ST-based implementation throughput approaches the theoretical initiation interval and can be characterized with about 2.7 μs latency. This is a result of different factors, e.g., DRAM controller delay, interconnect hierarchy and high-performance interface buffering logic. ST interface implementations provide higher throughput than any of the related work implementations, which is due to the pipelined nature of the accelerator.

Theoretical and practical measurements for the MM-based implementations are slower than implementations presented in [71, 72], although [71] does not provide a practical implementation. Of course, additional latency, in this case as well, is introduced by the interconnect

[12] http://git.edi.lv/rihards.novickis/generation_tool_hls_c_fully_connected_feed_forward_neural_network

Table 4.1. Small topology implementation resource utilization and benchmark result comparison table with [71], [72], [74], [75] (LUT-Look-up-Table, FF-Flip Flop, DSP-Digital Signal Processor core,BRAM-Block Random Access Memory).

Topology	Interface	LUTs	FFs	DSPs	BRAM	Target Initiation Interval	Theoretical Initiation Interval	Theoretical Latency	Latency		Throughput (Samples/s)	
									Original	Achieved	Original	Achieved
[71] 2-2-1	Memory-mapped	246 (0.46%)	118 (0.11%)	6 (2.73%)	3 (2.14%)	1	2	6	34 ns	555 ns $\sigma = 3.4$ ns	29,412,000	1,803,200
	Streaming	205 (0.39%)	135 (0.13%)	6 (2.73%)	3 (2.14%)	1	2	9		2.68 µs $\sigma = 37.5$ ns		49,272,000
[72] 2-4-1	Memory-mapped	980 (1.84%)	800 (0.75%)	48 (21.82%)	9 (6.43%)	1	2	10	44 ns	586 ns $\sigma = 3.4$ ns	2,272,700	1,705,600
	Streaming	983 (1.85%)	946 (2.92%)	48 (21.82%)	9 (6.43%)	1	2	12		2.69 µs $\sigma = 44$ ns		49,231,000
[74] 4-8-3-3	Memory-mapped	1304 (2.45%)	912 (0.86%)	0 (0.0%)	3.5 (2.5%)	1	4	12	1.16 µs	620 ns $\sigma = 8.7$ ns	862,070	1,612,400
	Streaming	1356 (2.55%)	1028 (0.97%)	0 (0.0%)	3.5 (2.5%)	1	4	19		2.78 µs $\sigma = 91$ ns		24,796,000
[75] 1-5-1	Memory-mapped	257 (0.5%)	78 (0.1%)	0 (0.0%)	1.5 (1.1%)	1	3	5	683 ns	578 ns $\sigma = 9.5$ ns	1,463,100	1,730,100
	Streaming	257 (0.5%)	81 (0.1%)	0 (0.0%)	1.5 (1.1%)	1	3	7		2.68 µs $\sigma = 35$ ns		33,005,000

hierarchy. Nevertheless, the presented approach performs better than [75]. The results in Table 4.1 suggest that the developed approach excels at maximizing NN throughput.

Although the topology presented in [76] implements the hyperbolic tangent function using Xilinx LOGICore IPs, the solution outperforms any of the versions presented in the paper, marking solution's suitability for a floating-point implementation. The reason for such an impressive performance difference is the distinct design goals. In [76], the author prioritizes on-the-fly reconfiguration of the network, while presented solution targets maximum throughput of the network.

The proposed approach still can be improved because it failed to achieve one cycle iteration interval for small FFNN topologies where it is certainly possible. Additionally, usage of HLS introduces additional logic, which was the main limitation to implementing more complex networks. Regarding the image processing pipeline in digital logic, although the AI-based methods present SoA performance, the usage of FFNN generally undermines the prospect of fully pipelined implementation.

The developed solution also has been applied for virtual sensor use-case, and some configurations achieved an impressive performance of two mega-samples per second, which even overcomes the results in the original article [8]. The approach potentially could outperform GPU implementations for virtual sensor use cases with a bigger FPGA chip while having the benefit

Table 4.2. Implementation resource use and benchmark result comparison table with [76] (Theoretical II = 100, Theoretical Latency = 987). (LUT-Look-up-Table, FF-Flip Flop, DSP-Digital Signal Processor core, BRAM-Block Random Access Memory)

Implementation	LUTs	FFs	DSPs	BRAM	Latency	Throughput (Samples/s)
[76]A	2232 (4.2%)	1210 (1.1%)	2 (0.9%)	-	33.1 ms	30.2
[76]B	3306 (6.2%)	1326 (1.3%)	4 (1.8%)	-	24.7 ms	40.5
[76]C	41,297 (77.6%)	33,395 (31.4%)	33 (15.0%)	-	5.7 ms	175.4
[76]D	51,028 (95.9%)	35,655 (33.5%)	65 (29.5%)	-	3.5 ms	285.7
Impl. Memory-Mapped	30,197 (56.8%)	55,231 (51.9%)	122 (55.5%)	6 (4.3%)	10.4 µs $\sigma = 0.011$	96,435
Impl. Streaming	31,246 (58.7%)	56,067 (52.7%)	122 (55.5%)	6 (4.3%)	12.5 µs $\sigma = 0.027$	997,852

of lower power consumption. The detailed virtual sensor performance comparison is provided in Appendix 10.

4.2 Heterogeneous System Architecture for Stereo Image Processing

A HSoC-based point-correspondence computational system has been developed – a challenge involving a range of design abstractions, i.e. digital circuit design, on-chip communications, Linux driver and system development and system architecture. The partitioned functional architecture of the designed solution is shown in Fig. 4.5. The processor (software) side of the HSoC ensures the overall control of the system and establishes communication with the off-board hardware (stereo camera through PCIe and demonstrator system through Ethernet) by utilizing the available software stack. Notably, the developed approach is also applied for implementing other computer vision algorithms[13].

The demonstrator integrates the following structure:

- ***Software components***

– *Acquisition component* (optional) orchestrates the system's image acquisition process. The software thread utilizes the *libdc1394* library for interfacing with the Bumblebee stereo camera and stores images in a memory region shared with the processing thread. The component

[13] Infrared image preprocessing in APPLAUSE project and vision-based odometry in COMP4DRONES project

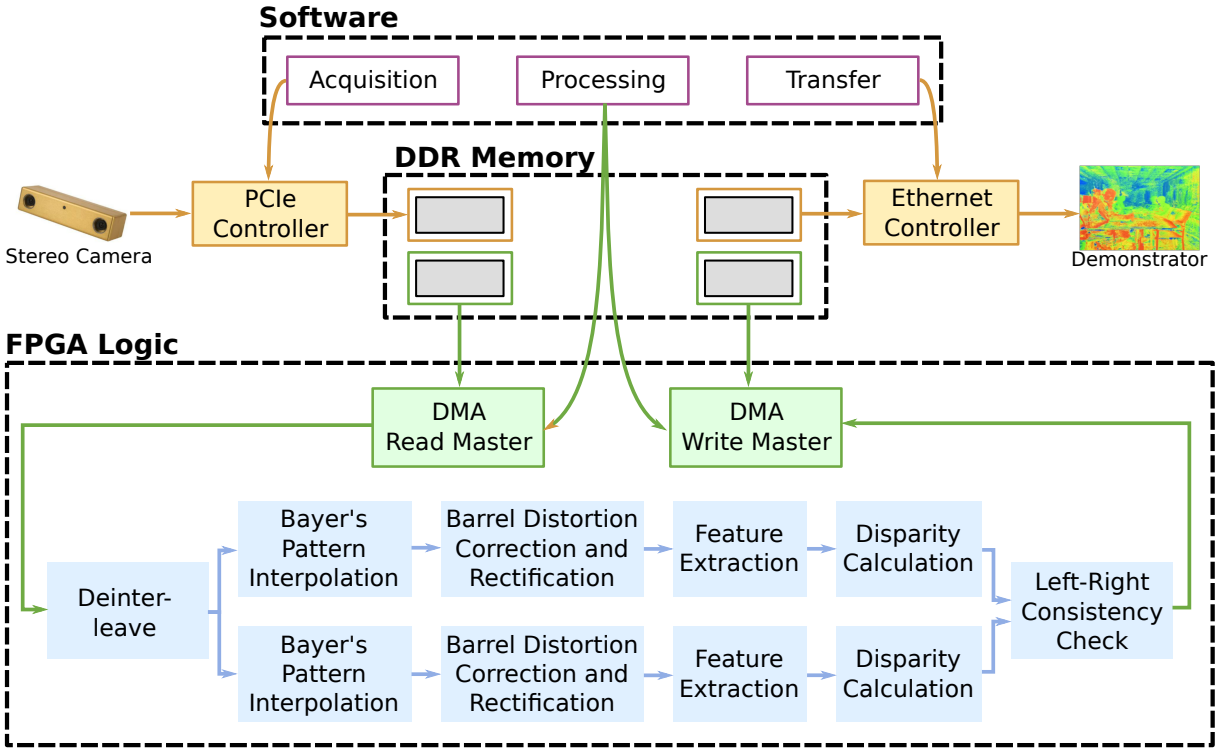


Figure 4.5. Functional architecture of the developed HSoC stereo-vision solution partitioned across processing paradigms and components.

can be considered optional because image acquisition also may be implemented in the programmable logic by ensuring direct interfaces to the cameras.

– *Transfer component* (optional) ensures the delivery of the processed image to the demonstrator display or next component in the processing pipeline. In the designed demonstrator, the component transfers processed images to the host computer, where an *Open Graphics Library* (OpenGL) application renders it to the display. Alternatively, this functionality also could be implemented in the programmable logic.

– *Processing component* ensures the accelerator's control and the actual processing in the programmable logic. This thread accounts for configuring the DMA transactions and the overall configuration and management of the accelerator pipeline.

• ***Programmable logic components***

– *DMA Write/Read Masters* are FPGA bus masters devoted to transferring images from the coherent (in respect to FPGA and MPU) memory to the acceleration pipeline and from the acceleration pipeline back to the coherent memory region. Current implementation utilizes proprietary DMA engines provided by Altera, now part of Intel [89].

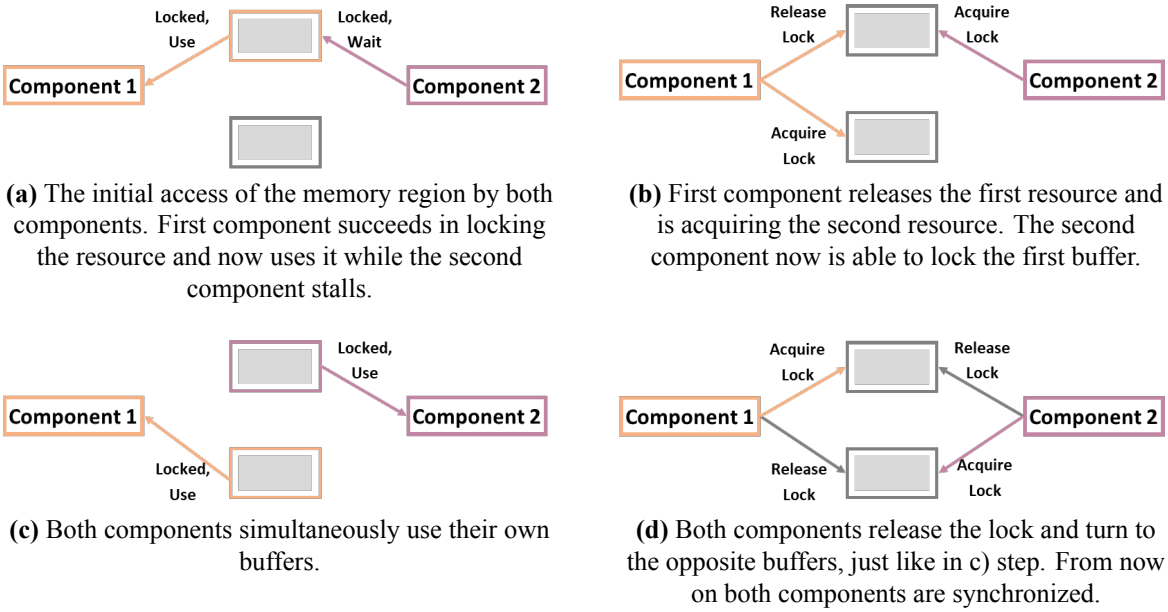
– *Deinterleaving* block addresses the characteristic of the particular stereo camera that produces

a synchronized yet interleaved image stream; therefore, the stream is deinterleaved into two separate image streams.

- *Bayer's Pattern Interpolation* deals with the underlying camera's image acquisition mechanism meaning that every pixel supplied by the Bumblebee camera is either red, green or blue, i.e. the user is exposed to the Bayer's pattern. The image needs to be interpolated [41]; in this particular case, the bi-linear interpolation algorithm is used.
- *Barrel Distortion Correction and Rectification* executes the spatial transformation of the separate image streams and therefore corrects the distortions introduced by the camera lenses and applies homography transformation to rectify the images and align epipolar lines with the horizontal axis.
- *Feature Extraction* characterizes every image pixel in terms of a descriptor – a vector containing a multitude of features. While feature sets can be modified to explore the design space and evaluate performance, the demonstrator system utilizes target pixel's and their neighbour intensity, horizontal gradient, vertical gradient and census transform.
- *Disparity Calculation* evaluates the actual correspondence that is found by comparing the feature vectors of the pixels in one image to the features of a single target pixel in the other. The SIP core estimates similarities by providing a confidence metric, and although the implementation of this block is relatively straightforward, it is very demanding in terms of resource consumption.
- *Left-Right Consistency Check* optionally performs filtering of outlier disparities by verifying that computed left and right disparity images correspond to each other, i.e. the computed correspondences are consistent across both images, meaning that they have a high probability of being estimated correctly.

The designed inter-communication mechanism utilizes shared coherent memory, as it enables other bus masters apart from the processor to perform memory transfer operations; nevertheless, there is a challenge of software component synchronization. The challenge has been solved by utilizing a double buffering technique [104], and Fig. 4.6 illustrates an example of a two-component synchronization procedure.

The developed system utilizes the double buffering technique two times - in between *Acquisition-Processing* and *Processing-Transfer* components. Two memory regions are shared and handled to ensure parallel component execution; for example, while the *Acquisition* thread



(a) The initial access of the memory region by both components. First component succeeds in locking the resource and now uses it while the second component stalls.

(b) First component releases the first resource and is acquiring the second resource. The second component now is able to lock the first buffer.

(c) Both components simultaneously use their own buffers.

(d) Both components release the lock and turn to the opposite buffers, just like in c) step. From now on both components are synchronized.

Figure 4.6. Software thread synchronization using double buffer technique.

writes input image into one of the buffers, the *Processing* thread uses the other buffer for configuring DMA transactions and transferring data through the accelerator. The same mechanism provides communication between the *Processing* and *Transfer* components; therefore, SoC simultaneously performs image acquisition, image processing and output image transfer resulting in a high-level pipeline. Notably, the *Cyclone V* SoC's processing system contains two cores, and only two processing units can truly execute simultaneously; nonetheless, the *Processing* component spends most of its time in a process queue as it waits for the end of DMA transactions. The system handles shared memory and controls DMA accelerators by using the same modules described in Section 3.1.

Further sections describe the actual digital implementations of the corresponding image processing algorithms.

4.3 Design of image processing accelerators

4.3.1 Deinterleaving of the input image stream

The first procedure implemented in the stereo-vision pipeline is the deinterleaving algorithm that splits the input image stream from the Bumblebee camera into multiple output streams. This requirement rises from the camera's synchronization characteristics as it avoids storing the entire image snapshots in-camera memory by interleaving data into a unified stream. Fig. 4.7

illustrates such input stream. Note that the used variant of the camera system incorporates three cameras.

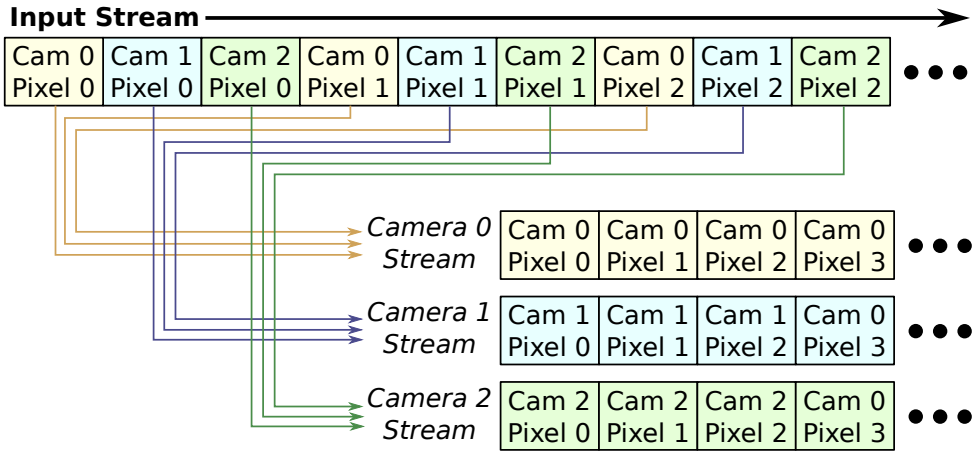


Figure 4.7. Interleaved input stream of the bumblebee camera.

The challenge is solved in the digital logic by using an of-the-shelf stream adapter and converting input stream width to 24 bits, i.e. resulting in 3 parallel pixels each expressed using a single byte. A further deinterleaving of the stream is relatively simple, as illustrated in Fig. 4.8. At this point, the image already resides in the DDR, and therefore there is no risk of the accelerator pipeline interfering with the Bumblebee camera streaming process. Nonetheless, directly interfaced cameras require additional FIFO buffers, as image sensors stream data assuming an always-ready data sink.

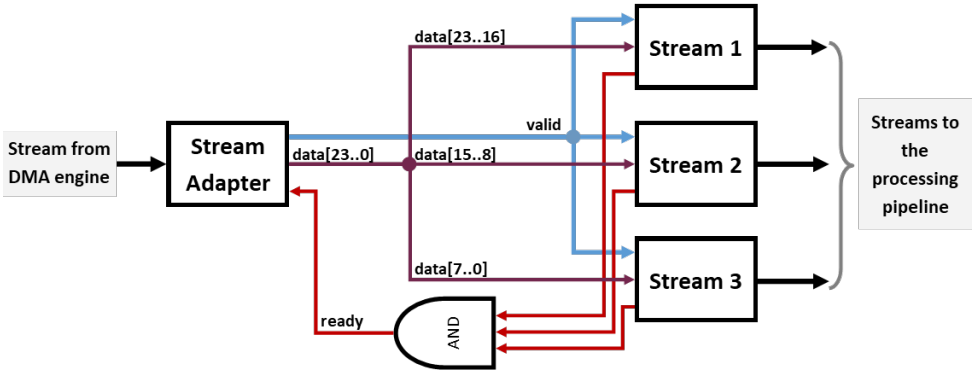


Figure 4.8. Separation of interleaved image data stream.

The synchronization aspect must be examined in more detail. Firstly, the solution is not perfect as the ready signals are ”anded” together. In longer processing pipelines, such an approach would lead to the degradation of the timing characteristics of the entire circuit. Nevertheless, such issues can be avoided by utilizing, for example, elastic buffers [113]. Secondly, some oper-

ations discussed further, e.g. rectification and lens distortion correction, have different latency characteristics for each of the sensors; therefore, every output stream should be buffered, .e.g. using FIFO buffers. Luckily, the relative differences can be calculated before the deployment of the processing pipeline, therefore ensuring suitable buffer size before synthesis.

4.3.2 Bayers pattern interpolation and RGB-to-Grayscale conversion

Modern cameras employ *Color Filter Array* (CFA), where different-colour filters reside on alternating sensor pixels. The most commonly used pattern in modern cameras is the *Bayer pattern* shown in Fig. 4.9, which places green filters over half of the sensors in a checkerboard pattern, while the red and blue filters cover the rest. The green pixels are also termed *luminance-sensitive elements*, while the red and blue ones – *chrominance-sensitive elements*. The number of green pixels is higher to mimic the physiology of the human eye, i.e. human eye's retina is most sensitive to the green light [114]. The process of *interpolating* the missing colour values to acquire valid RGB values for all pixels is known as *demosaicing*.

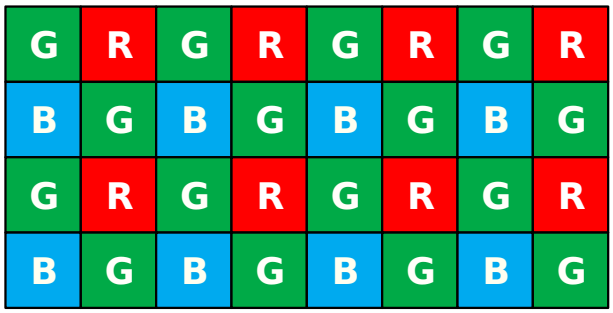
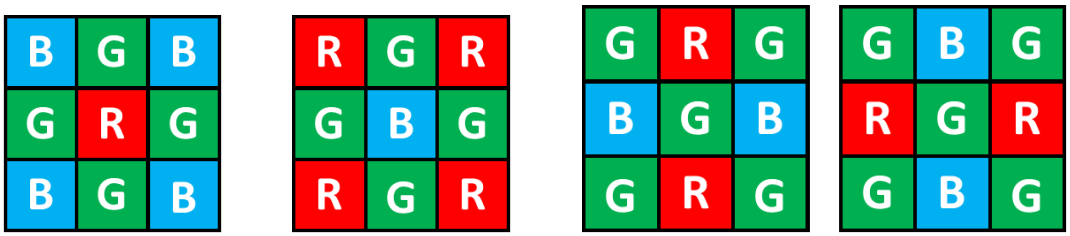


Figure 4.9. Bayer RGB pattern.

Although many demosaicing methods have been developed [115], the designed pipeline utilizes simple reconstruction based on linear and bilinear interpolation algorithms. Fig. 4.10 illustrates different patterns to be considered by the demosaicing algorithm.

On closer examination, it becomes evident that at any particular clock cycle, two colour values have to be inferred simultaneously. Further, the applied interpolation algorithm examines a region of 3×3 by using a sliding window approach, and at any particular clock cycle, either 4 (two and two) or 8 (four and four) values contribute to the reconstruction process. Fig. 4.11 represents the high-level structure of the designed circuit.

The input to the demosaic logic is provided by the sliding window block, which can be conveniently implemented in the digital logic [116]. The fully-pipelined demosaic block produces



(a) Red value is known, green and blue values are inferred from the adjacent pixels.

(b) Blue value is known, red and green values are inferred from the adjacent pixels.

(c) Green value is known, red and blue values are inferred from the adjacent pixels.

Figure 4.10. Bayer pattern variants considered for demosaicing algorithm.

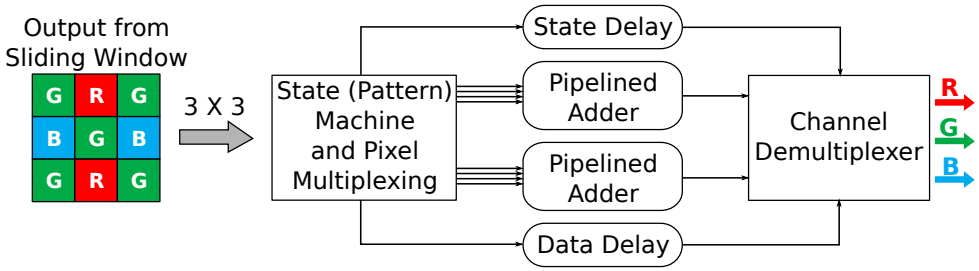


Figure 4.11. High-level structure of the designed demosaicing circuit.

sets of data corresponding to Fig. 4.10. Further, the proposed structure consists of a state generator, where each state corresponds to the patterns in Fig. 4.10. The block arranges these inputs for the pipelined adders, while the centre pixel is simply delayed because its value requires no reconstruction. The generated state is also delayed and fed into the demultiplexing block for output rearrangement. In cases when interpolation requires $2 + 2$ pixels, i.e. Fig. 4.10c, the same pixel pairs are provided to the pipelined adders twice. The outputs of the pipeline adders must be divided by 4, which is achieved by simply ignoring the two least significant bits.

The demosaic block also includes optional *RGB-to-Grayscale* conversion functionality as grayscale images reduce the number of operations triple-fold while not having major precision drawbacks, e.g. edge detection rate may be reduced by less than 10% [117, 118]. Therefore, a hardware-friendly *lightness* colourspace conversion method has been selected [119], which is expressed with the following equation:

$$\frac{max(I_R, I_G, I_B) + min(I_R, I_G, I_B)}{2}, \tag{4.4}$$

where I_R, I_G, I_B are the intensities of the red, green and blue pixels, respectively.

4.3.3 An approach to spatial image transformation

An essential part of any image pre-processing is the algorithms for pixel transformation or mapping, i.e. lens distortion correction, image rectification, digital zoom. Executing such tasks in digital logic is associated with higher complexity due to the semi-global nature of the memory access patterns. Notably, the storage of the entire image in the OCRAM memory of the programmable logic is either impossible (FPGA chips often have on-chip memory less than 1 MB) or expensive (a single dual-port *Static Random Access Memory* (SRAM) cell in the on-chip memory requires eight *Complementary Metal-Oxide Semiconductor* (CMOS) transistors, which results in a large area).

Notably, performing image transformations in digital logic is a significant field of research because data streaming and computer vision solutions require lower energy consumption, and edge devices more often include processing, e.g. specialized image processors emerge even within cameras. Nonetheless, the solution developed in this thesis challenges SoA as it formalizes the process to such an extent that a transformation circuit can be generated for any number of dimensions, not just images.

For example, Zemcík et al. [120] propose an efficient resampling algorithm based on separable *Finite Impulse Response* (FIR) filtering and bi-linear interpolation for geometry distortion correction, where distortion is described through a rectangular mesh. The pipelined solution separates vertical and horizontal resampling in independent modules with separate buffering schemes. While the solution is efficient, it is suitable for small geometrical distortions, where the displacement is only a few pixels.

Multiple teams [121] have developed solutions specifically for lens distortion correction. Clapa et al. [121] have optimized a fisheye distortion correction algorithm for hardware implementation using CORDIC [122] algorithms, while the actual distortion removal is managed by Microblaze soft processor.

Guo et al. [123] have developed a relevant system for the real-time image distortion correction based on a bilinear interpolation algorithm with a custom edge enhancement scheme. The algorithm sensibly utilizes four-block RAM buffering scheme for ensuring parallel access to the pixels. Notably, this thesis adopts a similar approach but extends it to even higher dimensions.

Another category of accelerators targets image scaling, where pixels are re-sampled at a dif-

ferent resolution and interpolation is necessary to restore the missing information. Aho et al. [124] presents a detailed look at a parallel memory unit describing separate address computation and data permutation blocks. Data element read and write addresses are computed with predetermined address functions, while the permutation block organizes data in a pattern based on the address of each element. A four point window for bilinear interpolation is achieved in this work, but increasing the neighbourhood size has not been demonstrated.

The proposed work is a generic solution for carrying out spatial image transformation in digital logic that enables real-time computing by omitting the need for introducing mechanisms for saving communication bandwidth as in [125] and enables fully digital implementation of such use case as real-time zoom lens distortion correction.

Fig. 4.12 illustrates the functional architecture of the spatial transformation accelerator that consists of input and output coordinate counters, dual-port memory matrix, writes and read masters, output buffering logic, control logic and external inverse transformation calculation logic.

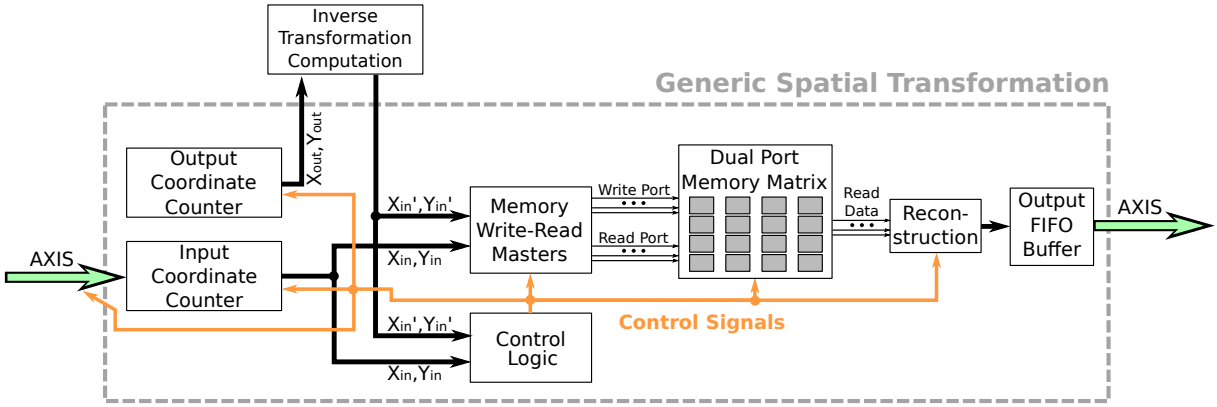


Figure 4.12. Approach to fully pipelined image transformation in digital logic.

Any spatial image transformation can be expressed as a mapping of input pixels to output:

$$x_{out}, y_{out} = f(x_{in}, y_{in}), \tag{4.5}$$

where x_{in}, y_{in} and x_{out}, y_{out} denote the input/output image coordinates, and f is some arbitrary function, often expressed as a matrix in the case of linear transformations. In such cases, incrementing input coordinates may result in ”hopping” for output coordinates. The proposed solution necessitates the opposite: computing the inverse transformation and essentially retriev-

ing the next input coordinate pair for the consecutive output coordinates, i.e.:

$$x_{in}, y_{in} = f^{-1}(x_{out}, y_{out}). \tag{4.6}$$

This functionality is achieved by the *Output Coordinate Counter* and *Inverse Transformation Computation* blocks. The *Inverse Transformation Computation* is external, it utilizes a simple ST interface and can incorporate any combination of spatial transformations, for example[14]:

$$x_{in}, y_{in} = f^{-1}_{\text{correction}}(f^{-1}_{\text{transformation}}(f^{-1}_{\text{zoom}}(x_{out}, y_{out}))). \tag{4.7}$$

Somewhat symmetrically, the *Input Coordinate Counter* provides image coordinates for the input data stream; therefore, the *Control Logic* has information about the data samples required by the transformation and the ones available in the buffers. This structure enables simultaneous processes of writing input data to the memory matrix and calculating the appropriate read addresses for the output data. Although, the circuit requires some time to initialize, which is determined by the first performed transformation. Notably, the dual-port memory matrix may be replaced with a single memory when using reconstruction based on the nearest-neighbour algorithm.

The designed circuit performs in the following manner: the *Inverse Transformation Computation* block receives the next necessary coordinates for the output image and calculates the ”required” coordinates for the pixel in the input image. Simultaneously, the *Input Coordinate Counter* generates coordinates for the received input image. The control logic compares both coordinate pairs, and if the ”required” coordinates are smaller than the input coordinates, it means that the input image still is not sufficiently buffered, the *Inverse Transformation Computation* and *Output Coordinate Counter* blocks are stalled until more of the input image data is available. If it appears that the ”required” pixels are loaded, then the transformation logic works simultaneously with retrieving the input image, i.e., the circuit operates in a fully pipelined manner.

The solution adopts a unique technique for signal reconstruction that enables N-point reconstruction while preserving memory resources. This technique is also generalized to any number of dimensions and is described in detail in the following Section: 4.3.4 *Parallel data access scheme for data reconstruction*. The tedious task of generating addresses for the individual memories in the *Dual Port Memory Matrix* is performed by the *Memory Write-Read Masters*.

[14]The f^{-1}_{zoom} and $f^{-1}_{transformation}$ are usually expressed as matrices and could be combined into a single matrix to save hardware resources.

In the particular use-case, the external *Inverse Transformation Computation* block corrects lens distortions and performs image rectification using homography matrix. The homography transformation is exected using a straightforward matrix multiplication; nonetheless, the homogeneous coordinates must be converted to cartesian. While the initial approach utilized a lookup table, which is based on the prior knowledge on the transformations, the current solution is more generic and utilizes Goldschmidt divider, which is also used in *Advanced Micro Devices* (AMD) processors [126, 127]. Fig. 4.13 illustrates the rectification circuit of the developed prototype system.

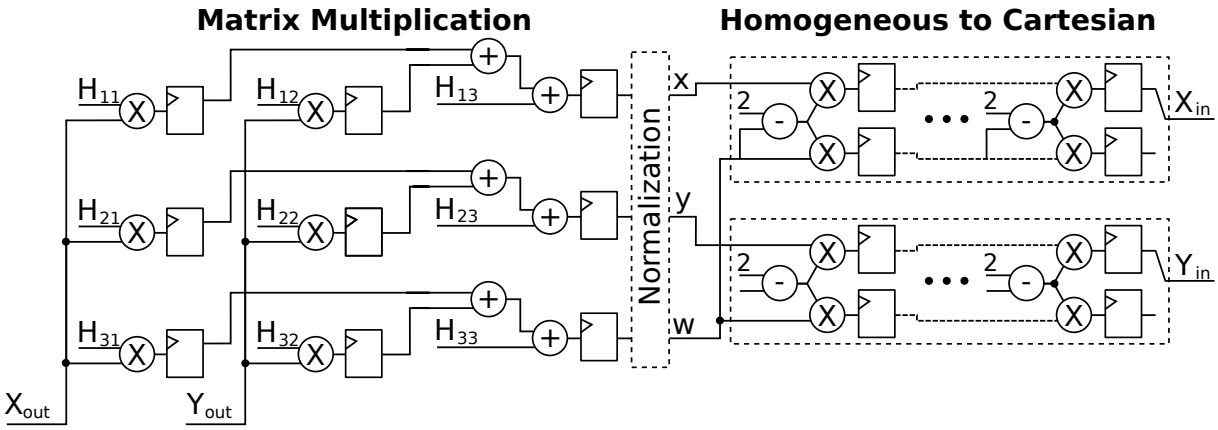


Figure 4.13. Representation of the digital circuit for homography transformation. Notably, the Goldschmidt divider requires the divisor to be in the range of $(0; 1]$; nonetheless, if the characteristics of the transformation are known beforehand and the computed scaler w is always in bounds, the normalization circuit may be omitted.

As for the lens distortions, the Bumblebee camera distorts images radially and the images can be corrected using the second-order approximation, i.e. by using the following set of equations:

$$\mathrm{r} = \left(\frac{\mathrm{X_{out}} - \mathrm{X_{center}}}{\alpha_\mathrm{x}}\right)^2 + \left(\frac{\mathrm{Y_{out}} - \mathrm{Y_{center}}}{\alpha_\mathrm{y}}\right)^2, \tag{4.8}$$

$$\mathrm{X_{in}} = (\mathrm{X_{out}} - \mathrm{X_{center}})(1 + \mathrm{K_1 r} + \mathrm{K_2 r^2}) + \mathrm{X_{center}}, \tag{4.9}$$

$$\mathrm{Y_{in}} = (\mathrm{Y_{out}} - \mathrm{Y_{center}})(1 + \mathrm{K_1 r} + \mathrm{K_2 r^2}) + \mathrm{Y_{center}}. \tag{4.10}$$

Fig. 4.14 represents the digital circuit executing the calculations of the Barrel distortion correction transformation, and Fig. 4.15 illustrates the transformation block's output with actual images from the Bumblebee camera. The designed transformation block is an essential part of the stereo-vision demonstrator system, as it cancels spatial image distortions and aligns epipolar lines.

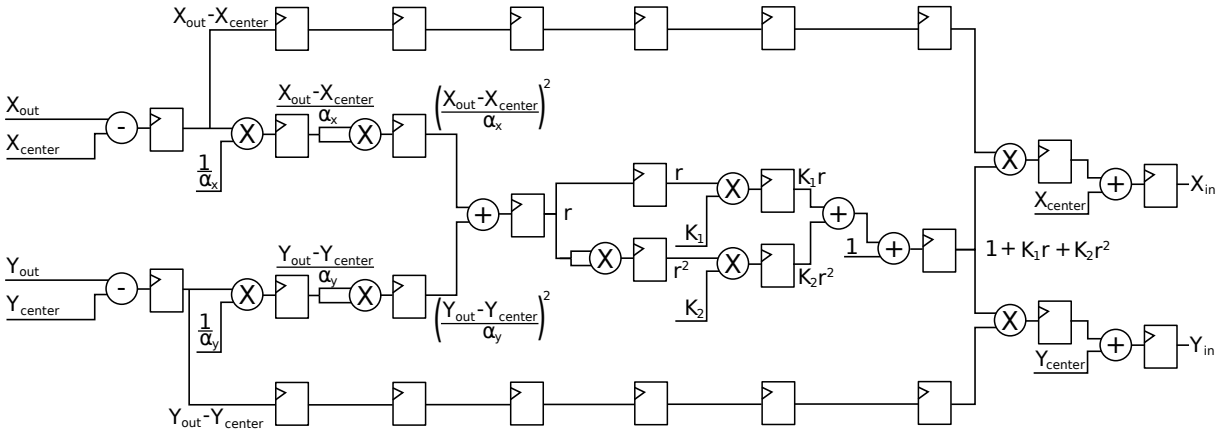


Figure 4.14. Representation of barrel (radial) distortion correction circuit for computing necessary input image coordinates.

Figure 4.15. Correction of radial lens distortions in Bumblebee camera using sparial transformation IP.

4.3.4 Parallel data access scheme for data reconstruction

Fig. 4.16 illustrates the overall approach to image buffering and data access. While the write pointer continuously rolls over, the read pointer updates simultaneously and is computed using the respective write and read requests. The approach is relatively straightforward with a single memory, i.e. the signal is reconstructed using the nearest neighbour interpolation method. Nonetheless, signal reconstruction using multiple samples introduces an additional complexity that yields a compelling optimization challenge.

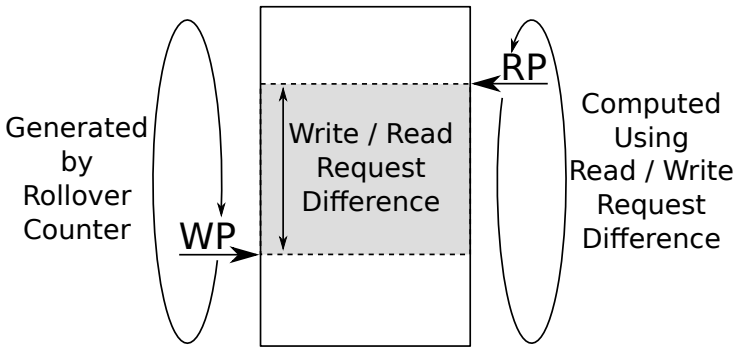


Figure 4.16. The overall concept of write and read pointer acess to the memory.

One of the principal requirements for carrying out a fully pipelined spatial transformation with interpolation is a memory buffering scheme that would permit interpolation logic to have simultaneous access to adjacent pixels. The simplest solution would be to use multiple dual-port memories as shown in Fig. 4.17. Although this construct is convenient, it can be optimised considering that pixel data for interpolation is adjacent. Fig. 4.18 illustrates a more efficient solution, where memories are reduced in size and only contain data for the respective (odd / pair) columns and rows. Although this solution reduces the required memory size four times, it depends on additional write and read logic for multiplexing.

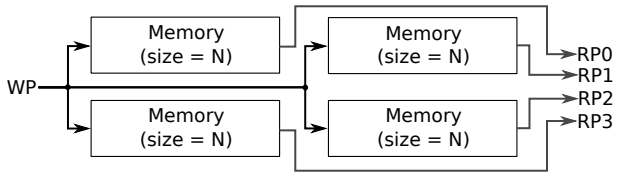


Figure 4.17. Simple data access scheme for 4-point reconstruction.

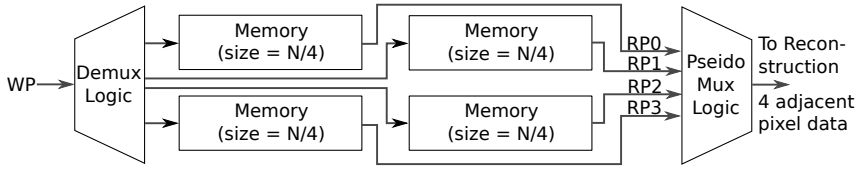


Figure 4.18. Optimized data access scheme for 4-point reconstruction with $4\times$ reduction in memory size.

Typically, the image address corresponds to the coordinates with the following equation:

$$a = Y \times W + X, \tag{4.11}$$

where a denotes address, W - image width, Y - image row and X image column. Nonetheless, the *Generic Spatial Transformation* circuit does not require storing the whole image in the

OCRAM. Utilizing the incrementing quality of the write pointer, we can substitute write address generation logic with a simple rollover counter with the maximum value corresponding to the size of the memory buffer. Nonetheless, we still require row and column signals for computing the difference between the write and read pointers.

Considering that the memory matrix consists of M rows and N columns, the last $log_2(M)$ row signal bits and $log_2(N)$ column signal bits control the demultiplexing logic for the write pointer's write request signal as illustrated in Fig. 4.19. Naturally, all write ports of the memory matrix share the write data and address signals.

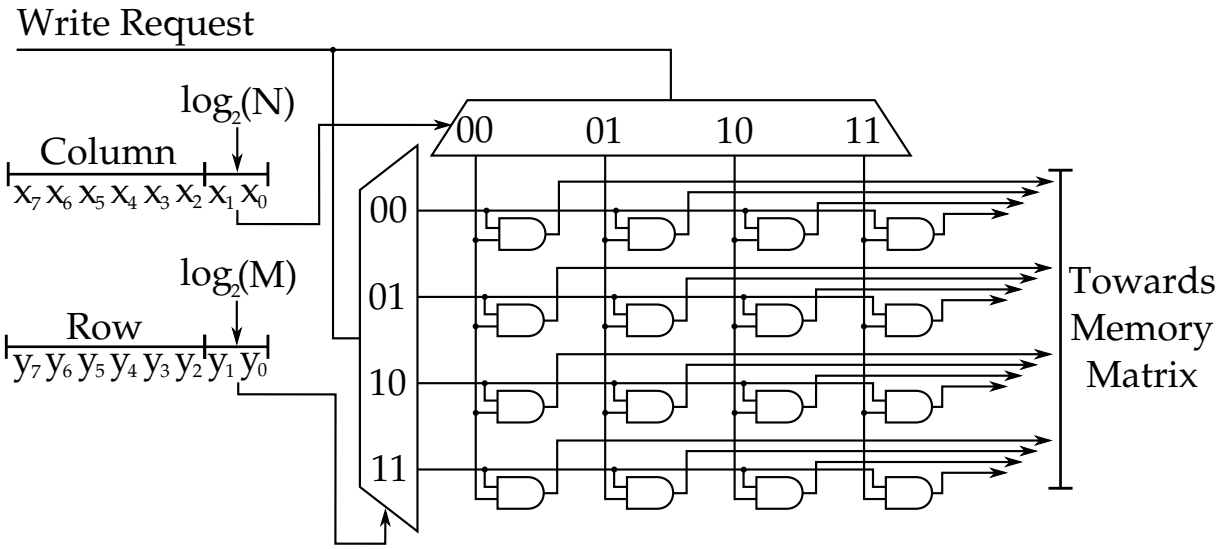


Figure 4.19. An example of write request demultiplexing logic when using 4×4 memory matrix.

Data retrieval is more challenging because there is a need of generating varied addresses for each of the memories' read ports. To formalize the derivation of such a circuit, first, let's assume a one-dimensional data reconstruction use-case using four samples, i.e. four memories. The Fig. 4.20 dissects a one dimensional request into r_f - fixed-point part later used for reconstruction, r_l - low bits used for individual memory address generation (dissected later) and r_h - high bits representing the core address for the sliced memories.

Fig. 4.21 illustrates an application of such request where stored data samples spread across four memories. Here i denotes the input sample index and a - memory address in the corresponding memory. The readout circuit can be considered an operator that takes the request and outputs the address vector for retrieving the relevant data. Recognize that while the address vector corresponds to the memories in order $m0-m3$, the retrieved data further should be reordered

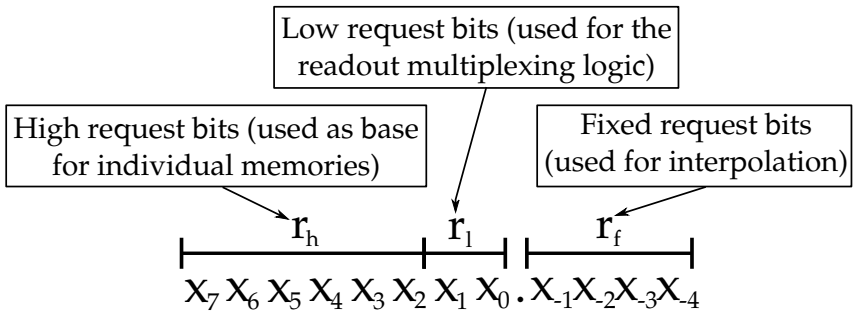


Figure 4.20. Dissection of a 12-bit reconstruction request for a 4-memory readout.

for sample reconstruction (interpolation).

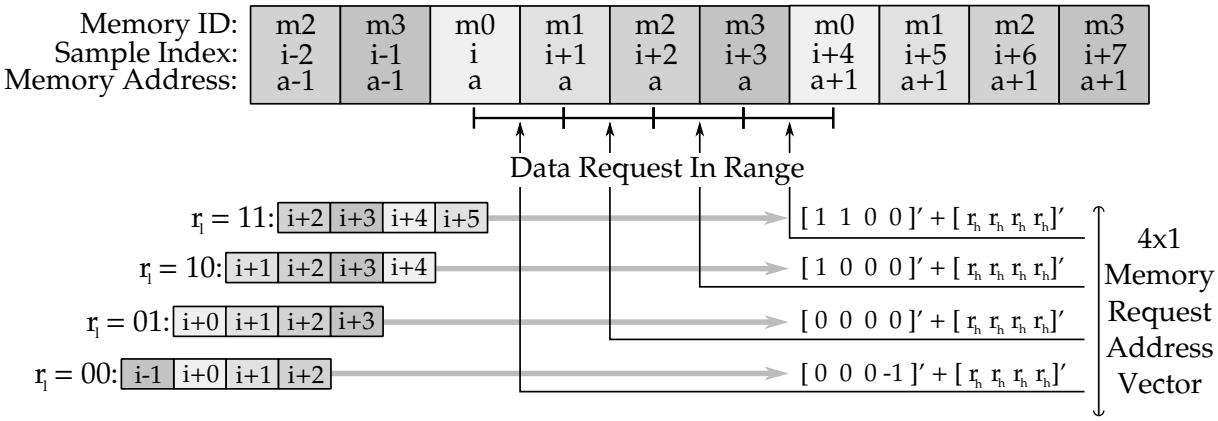


Figure 4.21. Data retrieval for reconstruction using four input data samples.

The illustrated data retrieval mechanism can be expressed in a matrix form using the following equation:

$$a = \mathbf{S}v_o + r_h, \tag{4.12}$$

where a denotes address vector, S - shift matrix, v_o - offset vector and r_h - high bits of the request. In this particular, case the offset vector would be:

$$v_o = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}' \tag{4.13}$$

and with J being the reverse identity matrix, the shift matrix is:

$$S_{4\times 7} = \begin{cases} \begin{bmatrix} 0_{4\times 3} & J_{4\times 4} \end{bmatrix}, \text{if } r_l = 0 \\ \begin{bmatrix} 0_{4\times 2} & J_{4\times 4} & 0_{4\times 1} \end{bmatrix}, \text{if } r_l = 1 \\ \begin{bmatrix} 0_{4\times 1} & J_{4\times 4} & 0_{4\times 2} \end{bmatrix}, \text{if } r_l = 2 \\ \begin{bmatrix} J_{4\times 4} & 0_{4\times 3} \end{bmatrix}, \text{if } r_l = 3 \end{cases} \tag{4.14}$$

Further, the offset vector can be constructed for any "power-of-two" number of memories - N - as:

$$v_o = [\underbrace{1 \cdots 1}_{\text{N/2}} \quad \underbrace{0 \cdots 0}_{\text{N}} \quad \underbrace{-1 \cdots -1}_{\text{N/2-1}}] \tag{4.15}$$

Let's advance the derived model by extending it to the second dimension by considering an unconventional signal reconstruction use-case where reconstruction is executed only across the vertical axis. Such constraints correspond to stretching or contracting the image only along the vertical axis. The Fig. 4.22 illustrates this use-case using four memories. The only difference compared to horizontal reconstruction is the offset added to the memory request address - it equals one-fourth of the image width, i.e.

$$C = \frac{\text{image width}}{log2(\text{number of memories})}. \tag{4.16}$$

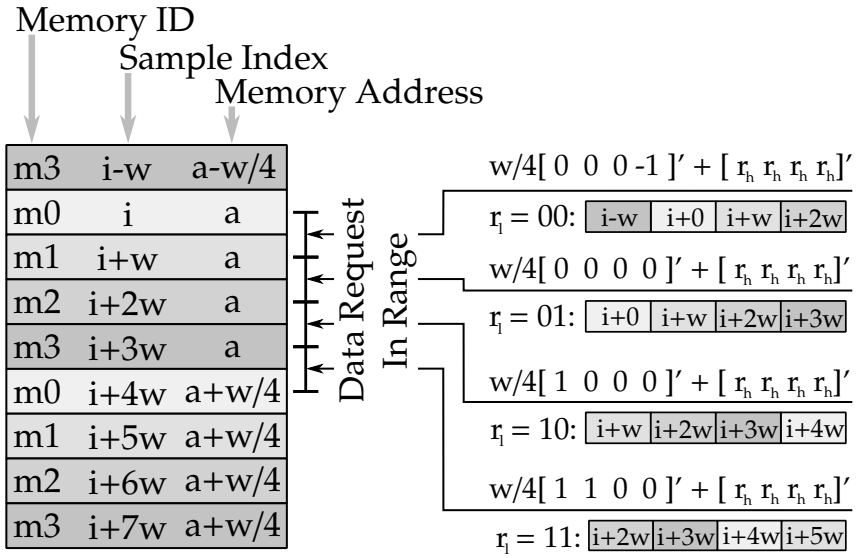


Figure 4.22. Data retrieval for reconstruction using four vertical input data samples.

The vertical retrieval mechanism can be expressed similarly to Eq. 4.12 by adding offset constant C:

$$a = C\mathbf{S}v_o + r_h, \tag{4.17}$$

Extending to any dimension, the address vector is:

$$a_n = C_n\mathbf{S_n}v_{o_n} + r_{h_n}, \tag{4.18}$$

where $C_1 = 1$ and n is dimension index.

Fig. 4.23 illustrates how the combination of vertical and horizontal retrieval mechanisms generate required addresses for all memories. The only difference here is that we have chosen to omit the offset constant from the offset vector to highlight the relationships between the combining output address matrix.

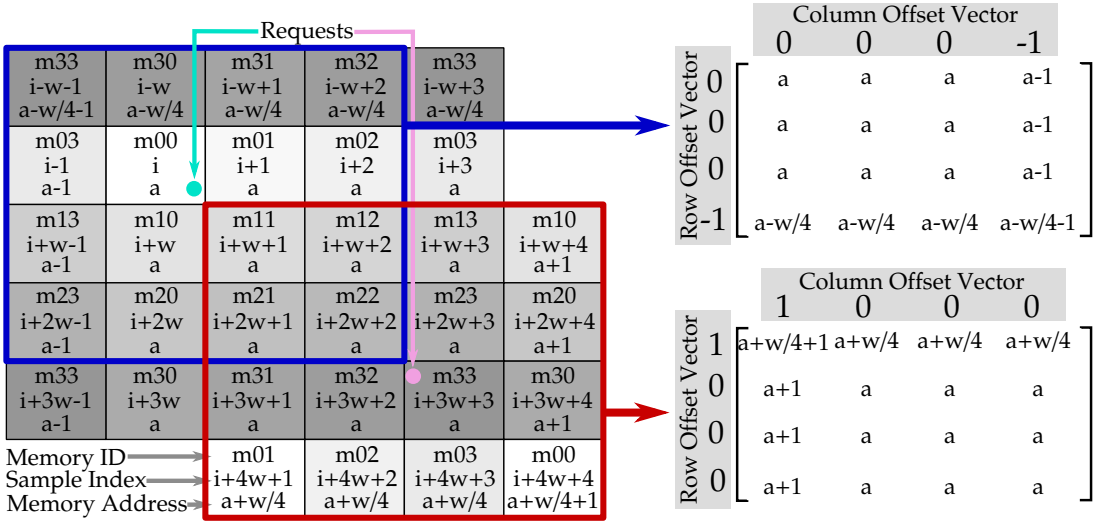


Figure 4.23. Address generation for 4x4 memory matrix.

Finally, this approach can be extended to any number of dimensions. For N dimensions memory addresses would be expressed as an N-dimensional matrix $A \in E^N$, where each element is computed by:

$$A_{i_1 i_2 \cdots i_N} = \sum_{n=1}^{N} C_n S_n v_{o_{i_n}}. \tag{4.19}$$

Further, the developed mathematical model can be mapped into particular digital circuits. Fig. 4.24 illustrates the overall data retrieval concept and the organizational structure of the necessary components. In the overall concept, there is an identical *Offset Vector* generation circuit for every dimension, but the *Read Access Generation* and *demultiplexing* circuits are unique for the particular dimensional variants. The inputs for the *Offset Vector* correspond to the r_l bits of the respective dimension, and *Read Access Generation* block must be also supplied with C_n constants, i.e. $C_1 = 1$, $C_2 = W/log2(N_2)$, $C_3 = H \times W/log2(N_3)$, etc. These constants are supplied externally to support variable input frame size, e.g. the constants could be routed from configuration registers as signal slicing replaces the apparent division.

Fig. 4.25 illustrates circuit for the offset vector generation and shifting, i.e. $S_n V_o$. The vector from the Eq. 4.15 is implemented statically, and the rest of the logic ensures the actual

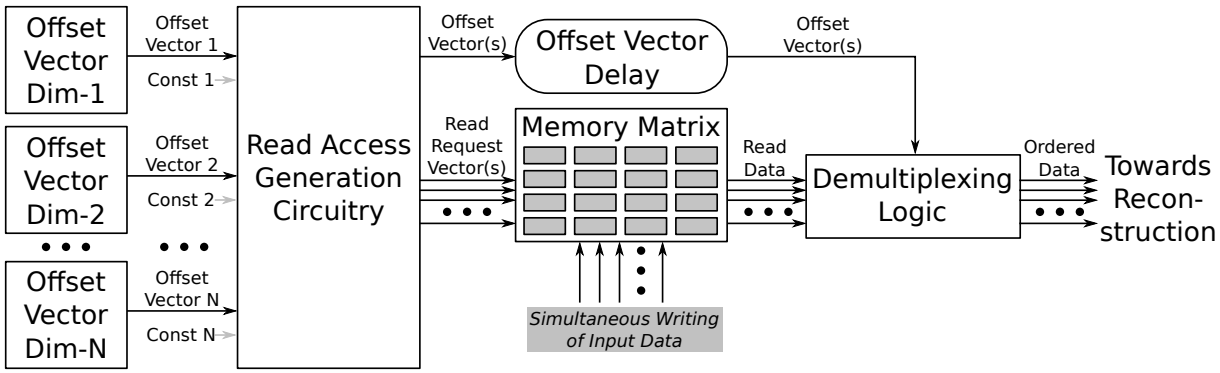


Figure 4.24. General address vector computing concept for N-dimensions.

shift operation. Notably, for larger vectors, i.e. higher number of memories, it would be more efficient to adopt a barrel shifter.

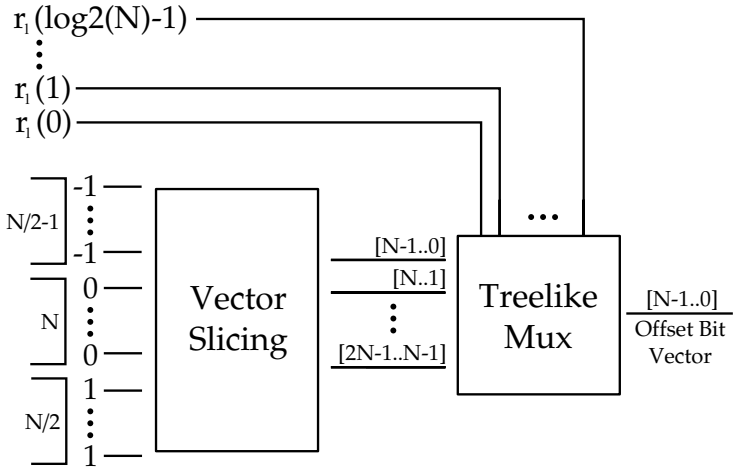


Figure 4.25. Generic offset vector construction circuit.

The read address generation is more complex and has to be implemented separately for each number of dimensions; nonetheless, patterns exist assisting such designs. Let's explore *Read Access Generation* logic for two dimensions, as shown in Fig. 4.26. The concept presents an approach consisting of three parts: computing all possible combinations for offset constants, computing all possible combinations for a memory address, multiplexing possible memory addresses to the actual memories. Firstly, the offset constant computation process is singled out because the constants do not change throughout the transformation, i.e. they must be updated whenever image dimensions change, meaning there is an opportunity for design optimization, as only a single adder can generate all the possible constants. Further, in the case of higher dimensionality, this constant matrix would transform into a constant cube, tesseract, etc. Next, these constants are added to the "current" read address, which is an operation that, in this partic-

ular case, requires eight adders for a fully pipelined design. At this point, all possible memory addresses are available, and they are routed accordingly for the memories. Fig. 4.26 illustrates such routing logic for the first column of the memory matrix. In a case of higher dimensionality, the routing logic is constructed similarly, but with an additional cascade of multiplexers for the particular plane of the memory address ”cube”.

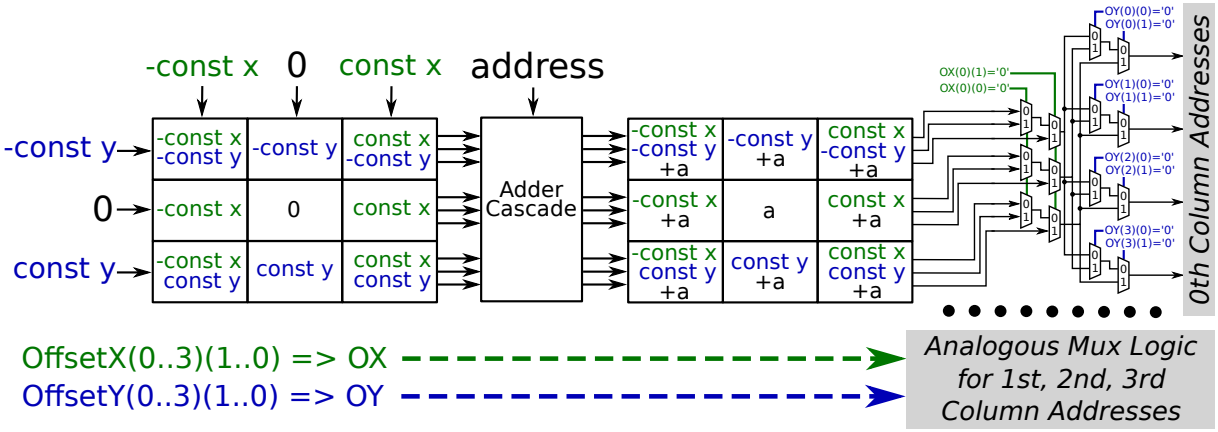


Figure 4.26. Conceptual deisgn of the read access circuitry.

Finally, the read memory data must be rearranged for a consistent reconstruction of the output sample. Fig. 4.27 illustrates such rearrangement logic for one-dimensional case that basically resembles the opposite action of the multiplexing logic in the offset generation circuit, i.e. Fig. 4.25. In a multi-dimensional case, the same circuit must be implemented for every dimension.

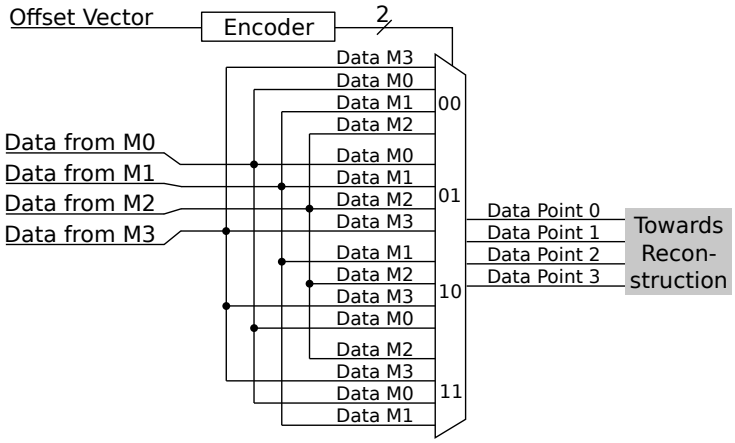


Figure 4.27. Conceptual deisgn data rearangment logic for 1-dimensional use-case.

Different memory access patterns have been examined to develop a general mathematical model that ensures simultaneous access to N dimensional data, marking the developed method applicable for the real-time reconstruction of volumetric data. The transformation and especially

the reconstruction logic is the most complex part of the designed stereo-vision image pipeline. The reconstruction method is not discussed further, and the reader is redirected to the work of Chiew et al. [128] who provide a comprehensive analysis of different image interpolation methods. The demonstrator system utilizes the nearest neighbour's reconstruction methods as the research for simultaneous N-dimensional data access came after the system's development.

4.3.5 Feature extraction

Feature extraction is one of the most suitable tasks for programmable logic when compared to other computing paradigms [129]. Nonetheless, feature comparison (pixel matching) in a fully pipelined application may require many adders, increasing the overall size of the circuit.

Notably, a hardware-efficient option - *census transform* - has been proposed by Zabih and Woodfill [130]. *Census* is a non-parametric transform $R_T(P)$ that maps the local neighbourhood surrounding a pixel P to a bit string representing the set of neighbouring pixels whose intensity is less than that of P. Let $\otimes$ denote concatenation, D – a set of displacements and let $\xi(P, P')$ be a transform that is 1 if $I(P') < I(P)$, then *census transform* can be specified as:

$$R_T(P) = \bigotimes_{[i,j] \in D} \xi(P, P + [i, j]). \tag{4.20}$$

Comparing two *census* vectors reduces to XOR operations, which makes it suitable for hardware implementation.

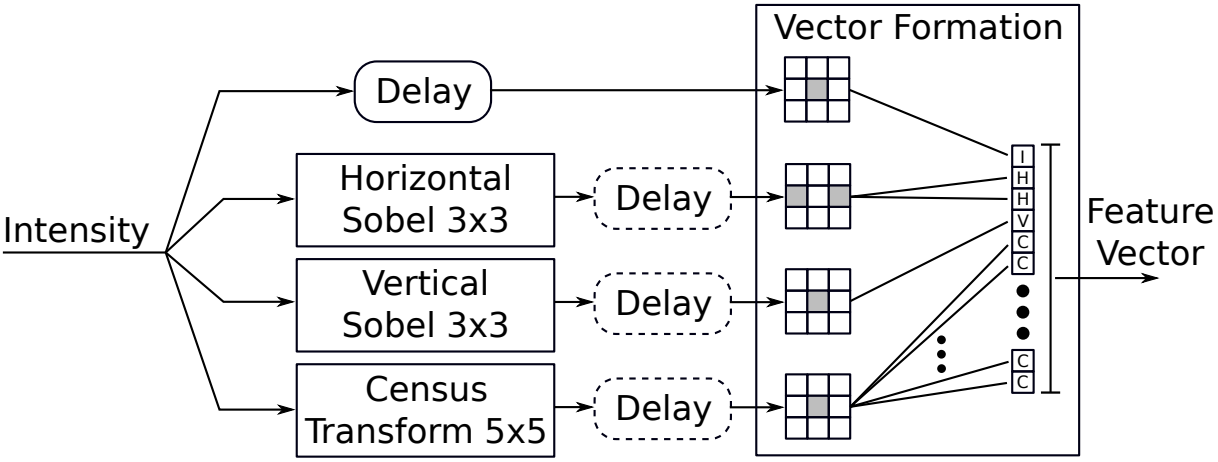


Figure 4.28. High-level composition of the implemented feature extractor.

Fig. 4.28 illustrates the feature extractor as implemented by the demonstrator system. The feature extractor provides four types of features: (1) pixel's intensity, (2) values of horizontal

Sobel filter for adjacent left / right pixels, (3) vertical Sobel filter at the pixel and (4) a census transform using 5x5 pixel region. For experimentation, the RTL has been developed using generics allowing for different feature configurations. Each feature also incorporates variable delay blocks, whose generation depends on the latency of the respective extraction block. The extraction itself is done in a fully pipelined manner using the sliding window approach. Finally, a vector from all the features is formed that further feeds into the correspondence matching components.

4.3.6 Correspondence calculations

Correspondence matching is the most resource consuming part of the whole stereo-image processing pipeline, because a fully pipelined implementation requires that all correspondence descriptor matching is accomplished in parallel. Fig. 4.29 illustrates the overall concept for simultaneous left-to-right and right-to-left correspondence matching using 128 points. The extracted feature descriptors are buffered using *Serial-In-Parallel-Out* (SIPO) buffers and linked through *Feature Comparator* blocks. The comparison results are further fed into *Correspondence Search Circuits* that identify a correspondence with the least amount of ”energy” (opposite of confidence). Notably, a single left image-based correspondence matching would be more efficient as 127 point descriptors would not be buffered.

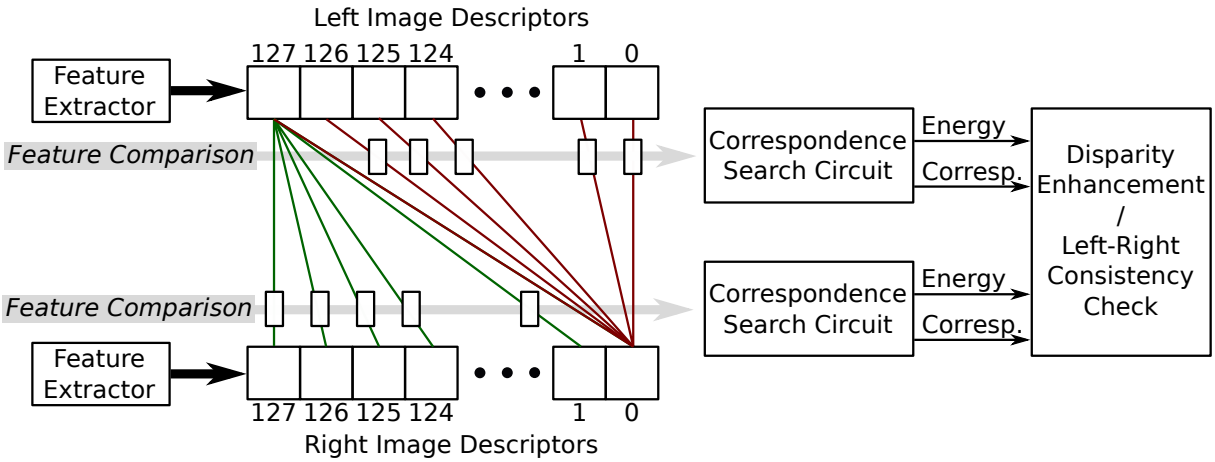


Figure 4.29. Composition of correspondence calculation and matching logic.

The feature descriptor comparison block is illustrated in Fig. 4.30, its goal is to provide a metric of point similarity, or in this case, dissimilarity. The comparison logic consists of several comparison blocks for comparing magnitudes of Sobel and intensity values and a block for

comparing census vectors, which is a simple bitwise XOR and bit summation. Essentially, the operations performed in this block are expensive as they should be multiplied with the number of correspondence matches in consideration, i.e. the demonstrator system considers 128 points.

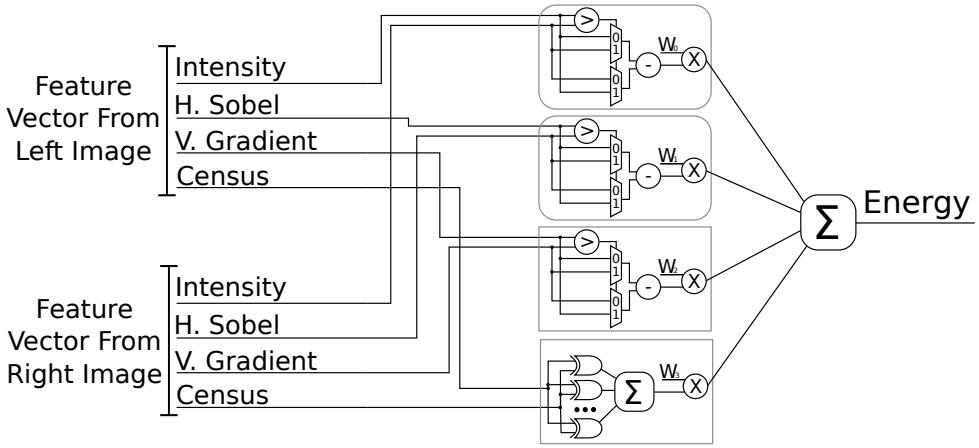


Figure 4.30. Feature descriptor comparison logic for a single pixel pair. Note that in the actual implementation, there are two features based on the horizontal Sobel values.

Fig. 4.31 illustrates the recursive (in terms of the RTL description) correspondence identification circuit. It accepts two arrays – correspondence point matching energies (disconfidence) and correspondence pixel indexes, which map directly to the distance of the correspondence pixel from the camera. As an output, the circuit produces the disconfidence of the match and correspondence index, i.e. disparity.

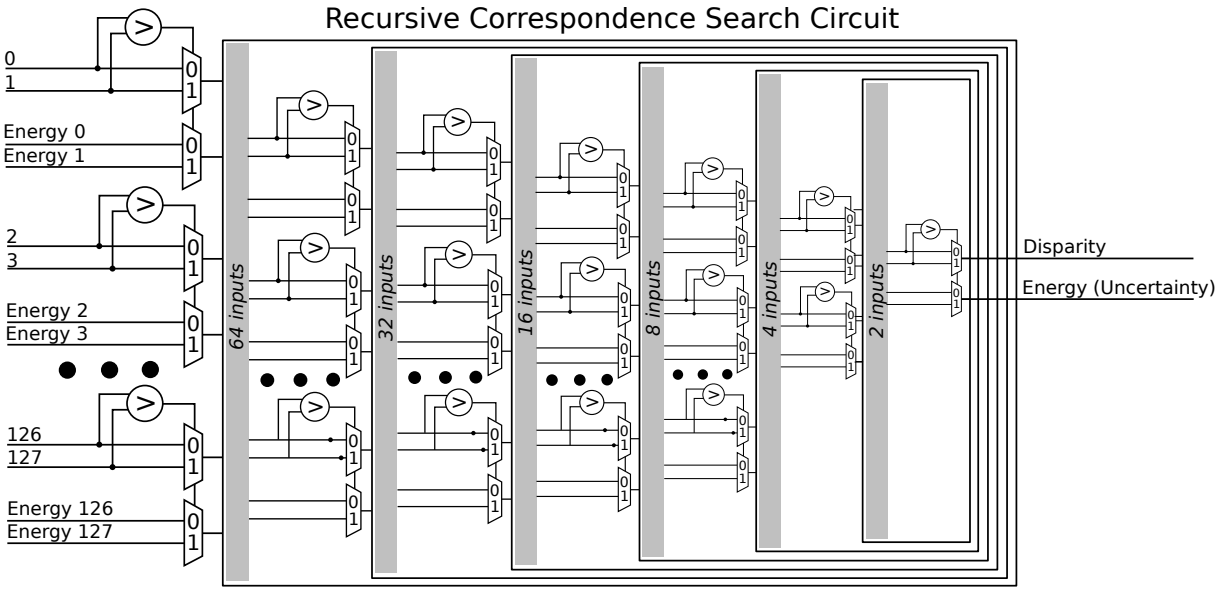


Figure 4.31. Correspondence identification circuit based on a recursive circuit description.

4.4 Demonstrator system and results

The demonstrator system has been developed according to the system architecture described in Section 4.2. Fig. 4.32 illustrates deployed demonstrator that is composed of Bumblebee BBX3 stereo-vision camera, Terasic's VEEK-MT Cyclone V heterogeneous SoC development kit and OpenGL-based host demonstrator. The developed technology targets high computational performance while having low-power consumption (< 10W) and overall costs. The SoC software application ensures image acquisition via PCIe, control of the FPGA accelerator pipeline and dispatching processed images to PC -based demonstrator via Ethernet. All disparity-related computations are carried out on FPGA logic (schematic described in *Very High Speed Integrated Circuit Hardware Description Language* (VHDL)), including an interpolation of Bayer filter mosaic, correction for barrel distortions, rectification, feature extraction and disparity calculation.

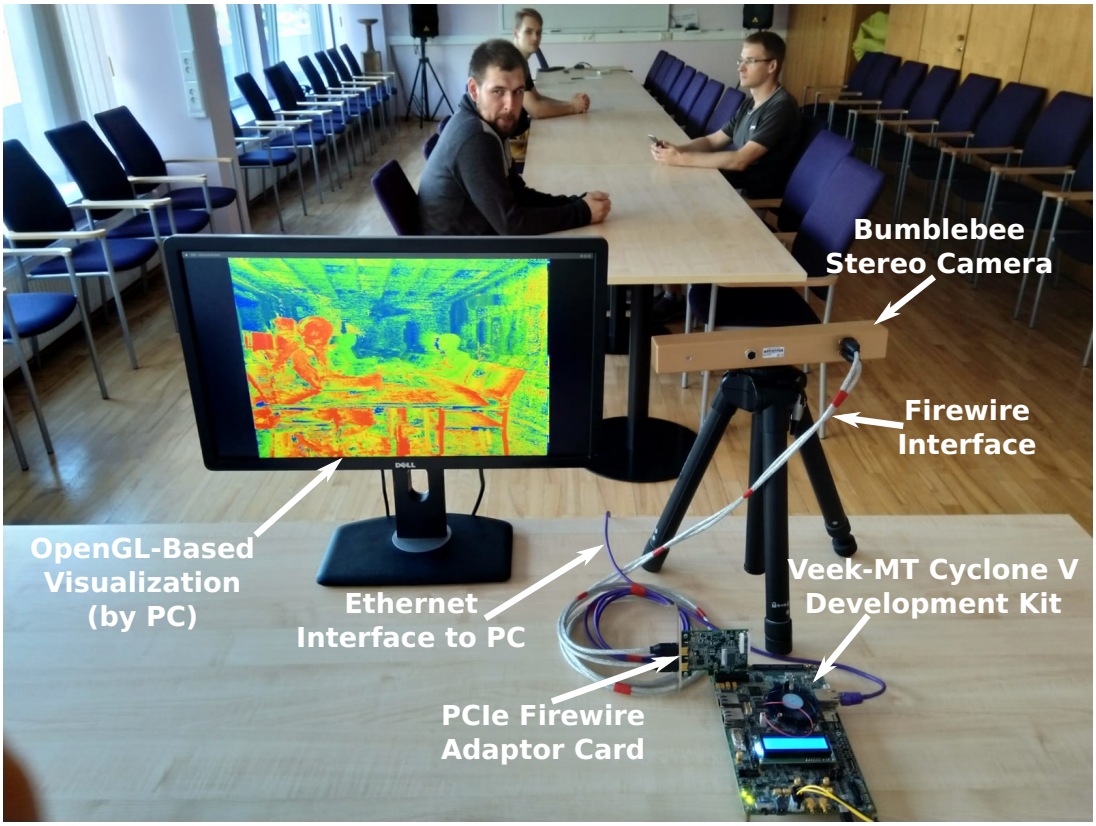


Figure 4.32. A demonstration of the stereo-vision demonstrator in action.

During the development of this thesis, Intel has disclosed an ASIC-based solutions, which has rapidly concured applications in field of robotics. Nonetheless, considering the sparsity of

the computed disparity map, the developed technology already can be used for depth sensing applications in power-critical systems such as mobile drones (obstacle avoidance) and some *Internet of Things* (IoT) use-cases. Fig. 4.33 illustrates the reconstructed 3D point cloud from the disparity image. While the reconstruction is performed using the OpenGL software, the necessary computations can be transferred into the digital logic if necessary.

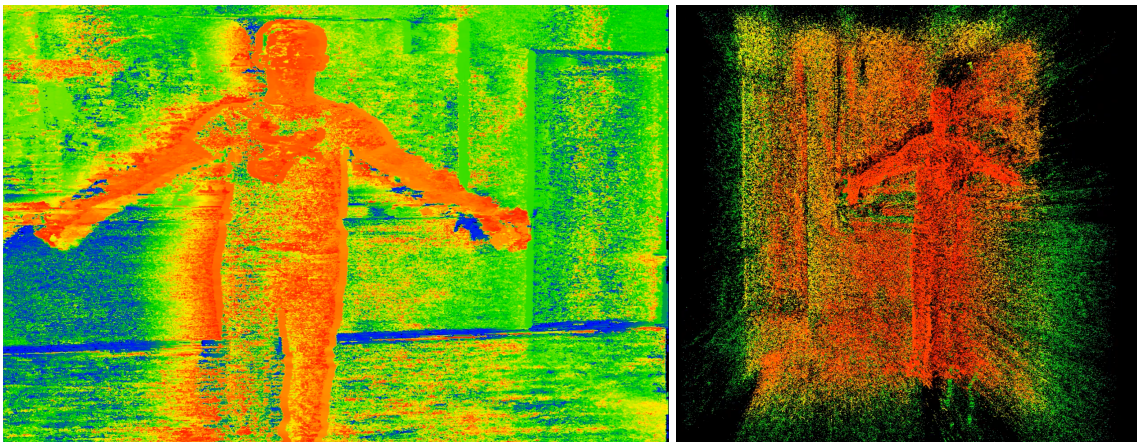


Figure 4.33. Demonstrator image 2D and 3D representations .

While the developed system's limitation is 16 fps due to the limitations imposed by the Bumblebee camera, the theoretical calculations (interface and FPGA accelerator throughput) suggest the maximum performance of at least 70 fps for the 1.4 MP resolution images using 100 MHz signal as the FPGA logic's clock.

5. CONCLUSIONS

This thesis addresses the relationship between computerized perception and increasingly complex HSoC technologies. In particular, the on-chip hardware and software co-architectures, implementation of stereo-vision and AI algorithms and associated real-time considerations. The primary aim of this thesis is to develop and improve computer vision development techniques and methods for HSoC technology. In order to achieve the set aim, five tasks were defined.

1. Identify methods for complementing RTL and software-based computing paradigms. This task was accomplished in Section 1 and Subsection 3.1. The literature review acknowledged the SoA in different computing paradigms that were further used to analyze their trade-offs. Further, this knowledge facilitated the development of architecture based on control by the MPU and offloading computing tasks to the FPGA. Both computing paradigms utilize memory-centric communication mechanisms for maximizing the overall throughput of the system.

2. Design heterogeneous architectures and tools for utilizing heterogeneous SoC technology. This task was accomplished in Section 3 by developing an FPGA-master based architecture for SoCs running Linux with the accompanying drivers and libraries for coherent contiguous memory management and DMA engine control. Further, an Asymmetric Multiprocessing (AMP) subsystem has been developed that enables real-time processing in Linux-based systems by offloading the critical applications to a fully dedicated bare-metal processing core. The developed solution unites the broad availability of open-source software with real-time control using a Linux driver interface. The driver ensures the setup of the bare-metal application and its configuration, AMP core's control, and it provides means for aggregating real-time performance characterization using *sysfs* interface. Finally, software-based system architecture implementation tools – *compage* and *icom* – have been developed. The tools enable the implementation of *blackboard* programming patterns for constrained Linux systems by giving the user means of configuring, replicating and interconnecting different software components, i.e. threads.

3. Design heterogeneous approach to the implementation of image processing pipelines. This task was accomplished in Subsections 3.1 and 4.2 by establishing a heterogeneous architecture for image processing pipelines using stereo-vision use-case as an example. The developed architecture utilizes software for image acquisition into coherent memory from a Bumblebee camera through PCIe, oversees the processing pipeline implemented in the FPGA and handoffs

the produced results to the demonstrator system via Ethernet interface. In the developed system, essentially, all components – software and hardware – execute simultaneously, thus achieving software + hardware parallelism.

4. Implement and conduct experimental research on the developed tools and algorithms. This task was accomplished in Section 4. First, an ANN-based solution was implemented in the FPGA hardware probing the possibility of a fully pipelined design. The developed approach is accompanied by a software tool for converting FFNN topologies into SIP cores with a streaming or memory-mapped interface. The achieved results not only outperform other solutions described in the literature but also show the applicability of the developed method for virtual sensor implementation, i.e. using the electric vehicle torque vectoring use-case as an example.

Further, a range of fully-pipelined accelerators have been designed for the implementation in digital logic, including deinterleaving of the combined input image streams, demosaicing of Bayer's RGB pattern, spatially transforming images to perform lens distortion correction and rectification, extracting features and computing the correspondence matching challenge. One of the main contributions is the generalization of the circuit for enabling fully-pipelined access to adjacent data samples for reconstruction in a memory-conserving way. While the developed solution is utilized for image processing, i.e. two dimensions, the generalized model enables the construction of a circuit for any number of dimensions; therefore enabling reconstruction, for example, of volumetric data. Finally, the developed accelerators were utilized for the development of the stereo-vision demonstrator.

5. Draw conclusions about the results of this Thesis. The main conclusion regarding the stereo-vision algorithm implementation using heterogeneous SoCs is that the unique blend of software and hardware ensures the feasibility of implementing such image processing pipelines. Furthermore, the technology achieves that in an energy-efficient way and provides extendability. The developed approach to image processing in heterogeneous SoC technology can (and is) applicable to other applications, e.g. processing of large (>50MP) images.

The value of the achieved results is further highlighted by several ongoing and finalized international research projects. For example, the software architecture implementation frameworks are being used for the development of AI-based perception system for vehicles (PRYSTINE, G.A. 783190, AI4CSM, G.A. 101007326) and control software for the control of autonomous drones (COMP4DRONES, G.A. 826610) while the developed image processing pipeline is

reused in the design of infrared image preprocessing algorithms (APPLAUSE, G.A. 826588).

Further, the outcomes of this thesis serve as a basis for an ongoing commercialization activity (SilHouse, No. KC-PI-2020/12) where a framework is being developed that joins a range of accelerators and provides a convenient software-based framework for their application to the industry.

APPENDICES

Architecture example: NXP i.MX 6Dual/6Quad

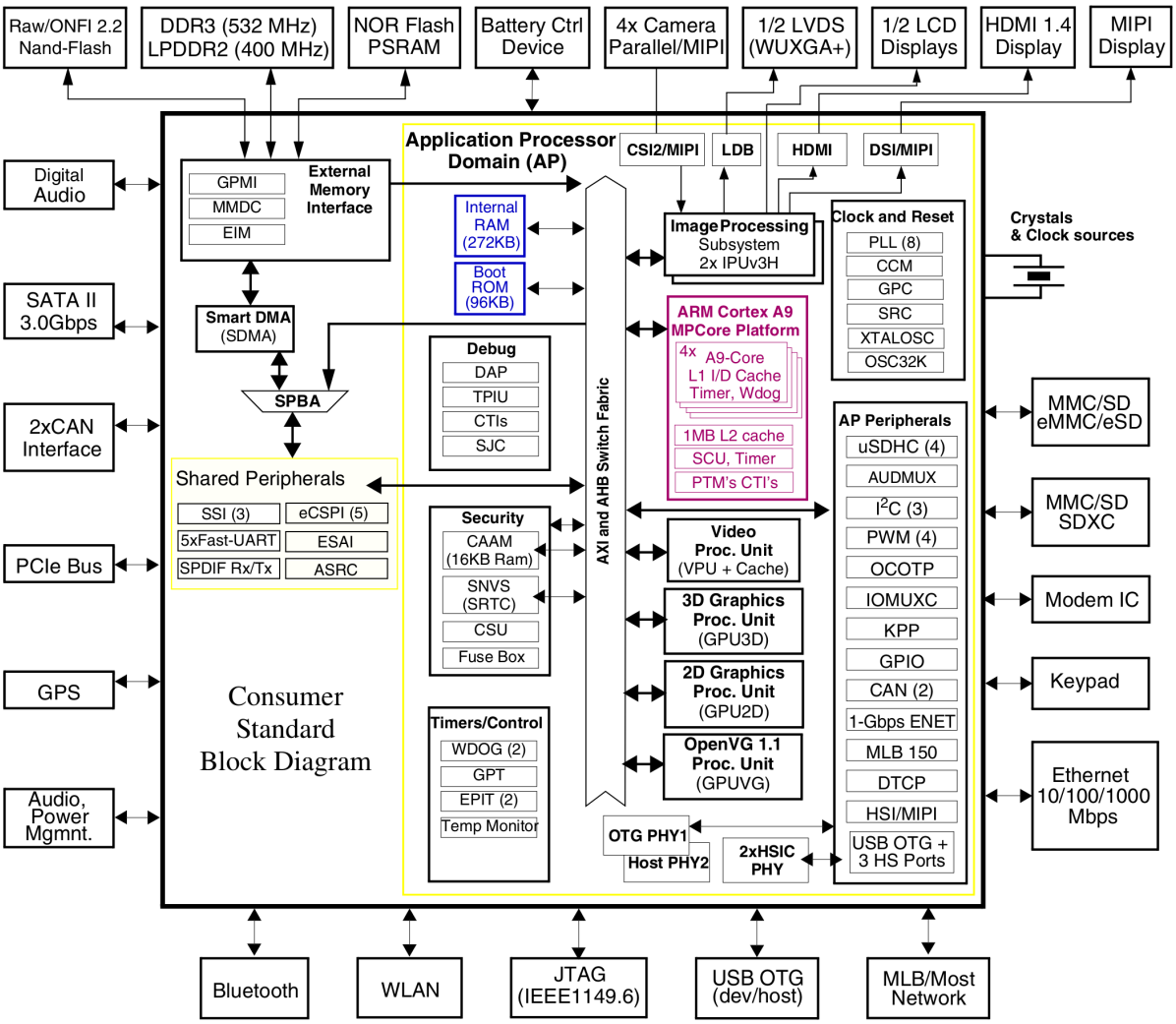


Figure A1. Block diagram of NXP's i.MX 6Dual/6Quad processor system [131].

Architecture example: Intel Cyclone V SoC

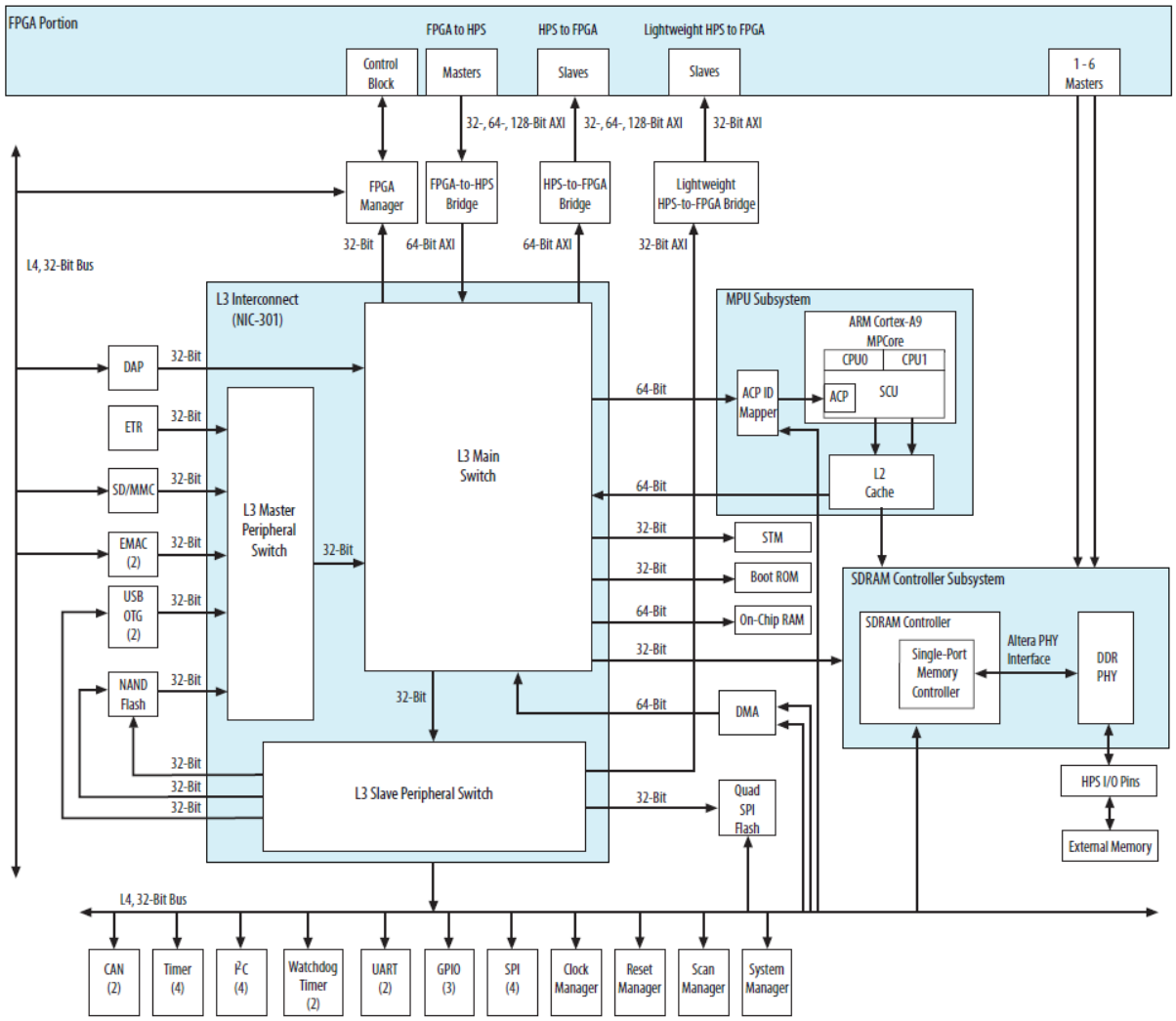


Figure A2. Block diagram of Intel's Cyclone V SoC [88].

FPGA Master-based communication throughput for Cyclone V SoC

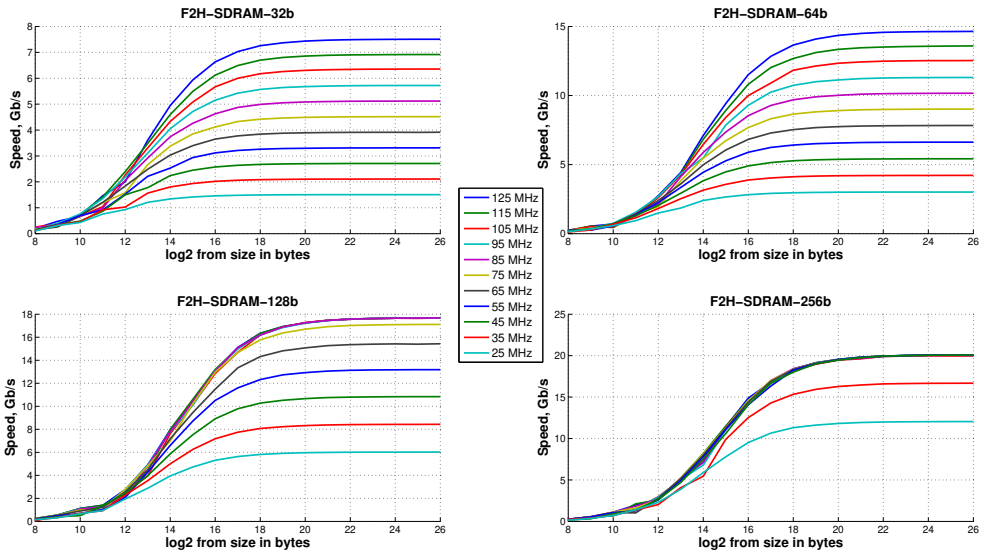


Figure A3. Master-based communication throughput measurements for Cyclone V SoC's data path Data path: FPGA-SDRAM.

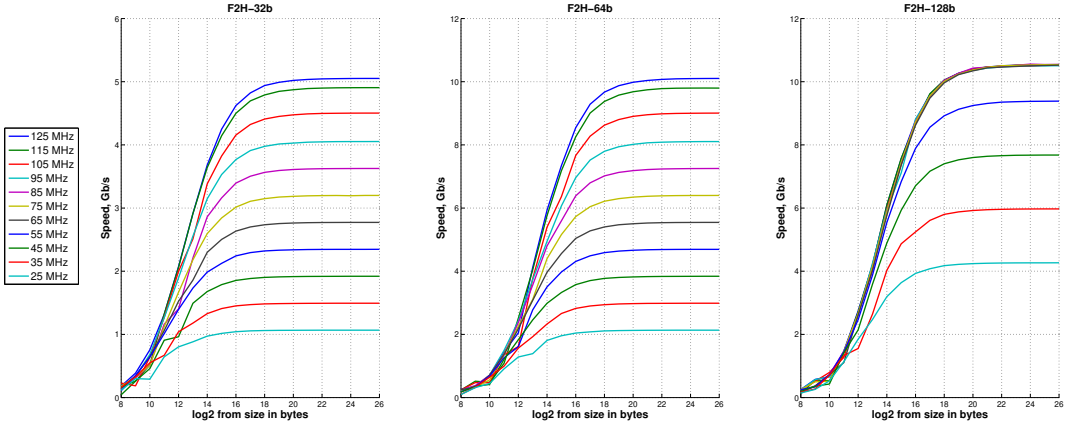


Figure A4. Master-based communication throughput measurements for Cyclone V SoC's data path Data path: FPGA-L3-SDRAM.

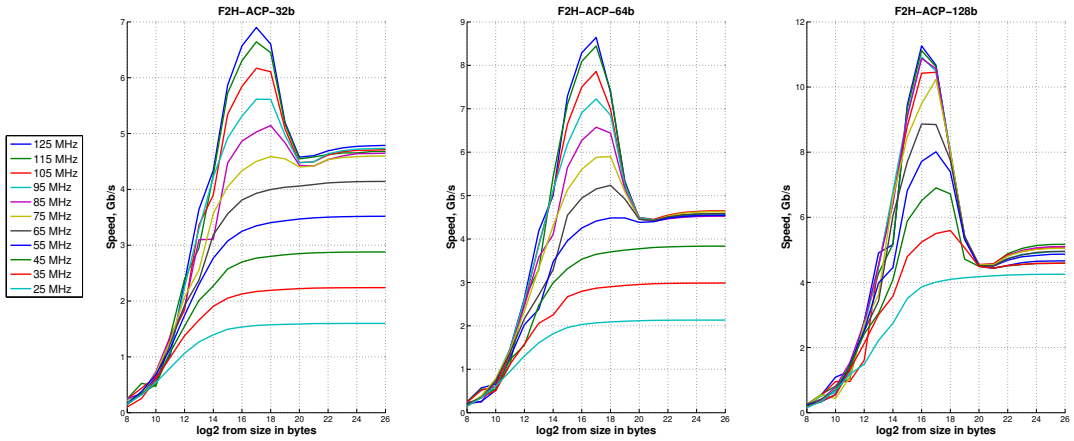


Figure A5. Master-based communication throughput measurements for Cyclone V SoC's data path Data path: FPGA-L3-ACP-SDRAM.

Table A1. Simultaneous read/write throughput measurement table

Frequency Scenario, [MHz]	F2H-S-32b, [Gbps]	F2H-S-64b, [Gbps]	F2H-S-128b, [Gbps]	F2H-S-256b, [Gbps]	F2H-L3-S-32b, [Gbps]	F2H-L3-S-64b, [Gbps]	F2H-L3-S-128b, [Gbps]	F2H-L3-ACP-32b, [Gbps]	F2H-L3-ACP-64b, [Gbps]	F2H-L3-ACP-128b, [Gbps]
20	1.20	2.41	4.82	9.64	0.85	1.71	3.41	1.28	1.71	3.41
25	1.51	3.01	6.02	12.04	1.07	2.13	4.27	1.60	2.13	4.25
30	1.81	3.61	7.23	14.43	1.28	2.56	5.12	1.92	2.56	4.87
35	2.11	4.22	8.43	16.67	1.49	2.99	5.97	2.24	2.99	5.60
40	2.41	4.82	9.64	18.78	1.71	3.41	6.83	2.56	3.41	6.21
45	2.71	5.42	10.84	**20.08**	1.92	3.84	7.68	2.88	3.84	6.90
50	3.01	6.02	12.02	20.07	2.13	4.27	8.53	3.20	4.25	7.54
55	3.31	6.62	13.18	20.08	2.35	4.69	9.38	3.52	4.55	8.01
60	3.61	7.23	14.32	20.07	2.56	5.12	10.20	3.84	4.88	8.71
65	3.91	7.83	15.44	20.08	2.77	5.55	**10.52**	4.14	5.24	8.86
70	4.22	8.43	16.54	20.08	2.99	5.97	10.52	4.39	5.57	9.49
75	4.52	9.02	17.11	20.07	3.20	6.40	10.54	4.60	5.90	10.24
80	4.82	9.62	**17.58**	20.05	3.41	6.83	10.55	4.86	6.23	10.49
85	5.12	10.16	17.68	20.06	3.63	7.25	10.56	5.14	6.57	10.88
90	5.42	10.72	17.68	20.08	3.84	7.68	10.53	5.40	6.96	11.13
95	5.72	11.31	17.68	20.08	4.05	8.11	10.51	5.61	7.23	10.92
100	6.02	11.89	17.68	20.08	4.27	8.53	10.52	5.88	7.48	11.10
105	6.36	12.53	17.68	20.08	4.50	9.01	10.52	6.17	7.86	10.45
110	6.62	13.00	17.68	20.08	4.69	9.38	10.52	6.39	8.16	11.00
115	6.92	13.59	17.68	20.08	4.91	9.80	10.53	6.64	8.45	11.12
120	7.20	14.11	17.68	20.08	**5.05**	**10.08**	10.54	6.80	8.62	11.19
125	7.51	14.64	17.68	20.08	5.05	10.10	10.54	6.90	8.64	11.26

High-level architecture of Xilinx Zynq Ultrascale+ MPSoC

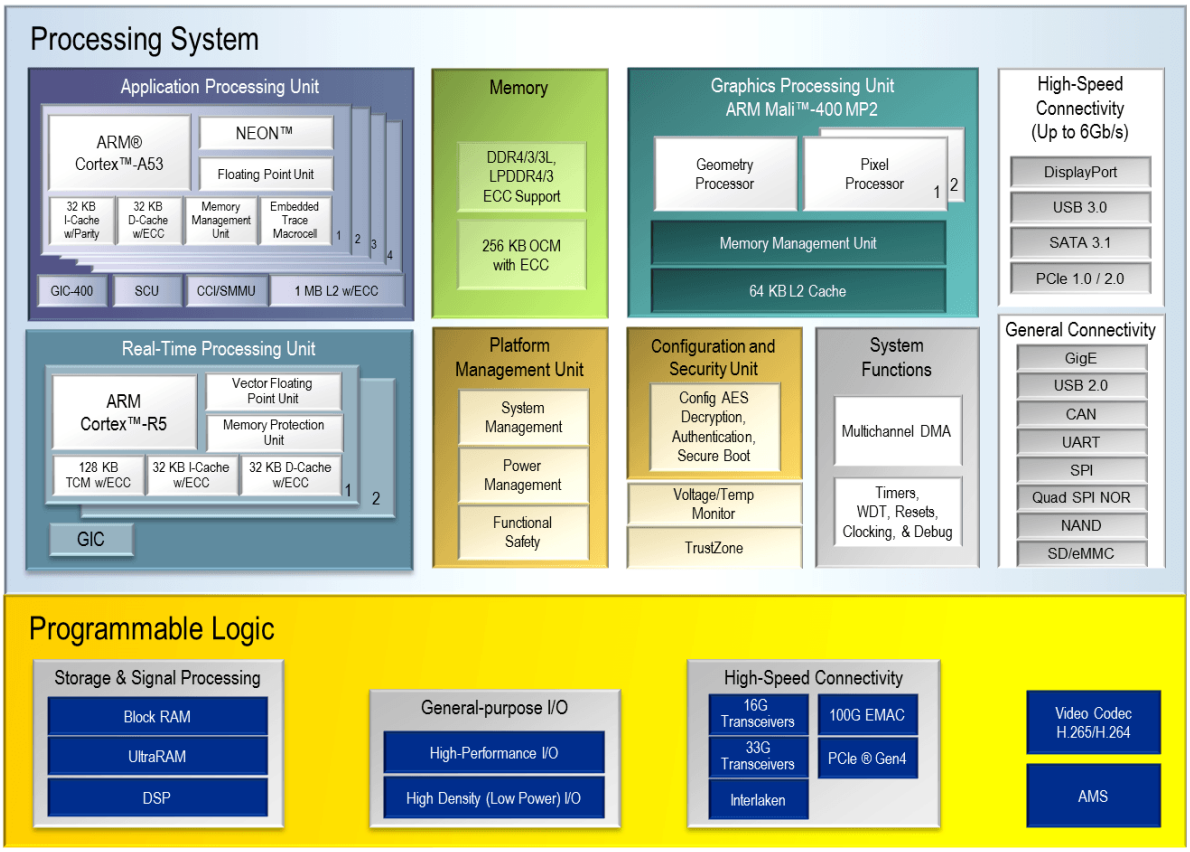


Figure A6. High-level architecture of Xilinx Zynq MPSoC Ultrascale+ SoC [132].

A simplified view of the Avalon Memory Mapped interface.

Table A2. Simplified list of Avalon-Memory Mapped interface.

Signal	Width	Direction	Description
address	1-64	Master->Slave	Signal represents a byte address. (By default, the interconnect translates the byte address into a work address)
byteenable *byteenable_n*	2,4,8,16,32, 64,128	Master->Slave	Enables one or more specific byte lanes during transfers on interfaces of width greater than 8 bits. Each bit corresponds to a byte in writedata and readdata.
read *read_n*	1	Master->Slave	Asserted to indicate a read transfer. If present, readdata is required.
readdata	8,16,32,64, 128,256, 512,1024	Slave->Master	Data driven from the slave to the master in response to a read transfer.
response	2	Slave->Master	An optional signal that carries the response status. 00: OKAY 01: RESERVED 10: SLAVEERROR 11: DECODEERROR
write *write_n*	1	Master->Slave	Asserted to indicate a write transfer. If present, write-data is required.
writedata	8,16,32,64, 128,256, 512,1024	Master->Slave	Data for write transfers.
lock	1	Master->Slave	Ensures that once a master wins arbitration, the winning master maintains access to the slave for multiple transactions.
waitrequest *waitrequest_n*	1	Slave->Master	Asserted when slave is unable to respond to a read or write request.
readdatavalid *readdatavalid_n*	1	Slave->Master	When asserted, indicates that the readdata signal contains valid data.
writeresponsevalid	1	Slave->Master	When asserted, the value on the response signal is a valid write response.

Appendix 6

Example of *compage* framework's instantiation

```
#include "compage.h"

int main(int argc, char *argv[]){
    return compage_main(argc, argv);
}
```

Appendix 7

Example of *compage* component's description

```
/* 1. include compage header file */
#include "compage.h"

/* 2. define component's data structure */
typedef struct {
    uint32_t    a;
    float       b;
    char        *c;
} pdata_t;

/* 3. provide default component's data structure  */
pdata_t pdataDefault = {
    .a = 3735928559,
    .b = 3.14159265,
    .c = "String parameter";
};

/* 4. describe the component  */
void *componentName(void *p){
    pdata_t *pdata = (pdata_t*)p;

    /* implementation of the component */
}

/* 5. register component and configuration data with the framework  */
COMPAGE_REGISTER(componentName, pdataDefault);
COMPAGE_PDATA_ADD_CONFIG(componentName, pdata_t, a);
COMPAGE_PDATA_ADD_CONFIG(componentName, pdata_t, b);
COMPAGE_PDATA_ADD_CONFIG(componentName, pdata_t, c);
```

Appendix 8

Example of *compage* framework's configuration file

```
# first instance of the component
[component_0]
handler=componentName
enabled=1
a=0
b=2.7182
c="Updated string parameter"

# another instance of the component
[component_1]
handler=componentName
enabled=1
a=0
b=2.7182
c="Another updated string parameter "

# 3rd instance of the component, but it is disabled
[component_2]
handler=componentName
enabled=0
a=0
b=2.7182
c="Yet another updated string parameter "
```

Abstract example of *icom* framework's usage

```
void *componentPull(void *arg){
    /* 1. Initialize communication */
    icom_t *icom = icom_initPull("inproc://push_pull", PAYLOAD_SIZE, flags);
    char *buffer;

    /* 2. Communicate and get buffer */
    ICOM_DO_AND_FOR_EACH_BUFFER(icom, buffer);

    /* 3. do work */

    /* Receive buffers - END */
    ICOM_FOR_EACH_END;
}

void *componentPush(void *arg){
    /* 1. Initialize communication */
    icom_t *icom = icom_initPush((char*)arg, PAYLOAD_SIZE, 2, flags);
    char *buffer;

    /* 2. Get buffer */
    ICOM_GET_BUFFER(icom, buffer);

    /* 3. do work */

    /* 4. Communicate */
    ICOM_DO(icom);
}
```

Performance metrics of the developed FFNN implementation approach targeting virtual sensor use case, presented by Dendaluce et al.

Table A3. FPGA resource utilization and performance metrics for 8-16-12-8-4 FFNN [8]. Data type: single floating point; Activation function: implemented using LogiCore IP. (Theoretical throughput is provided for Streaming implementation.)

NN Data Type	Act. Func.	Iter. Interv.	Resource Utilization								Latency (µs)	Throughput (Samples/s)	Absolute Mean Error
			BRAM		DSP		Registers		LUTs				
			Tot.	%	Tot.	%	Tot.	%	Tot.	%			
Fixed 14, 5	LUT 8, 1	32	8	20.79%	14	5.71%	22452	21.1%	8958	16.84%	10.221 $\sigma = 0.03$ µs	1.52 M $\sigma = 0.046$ µs	0.0232
Fixed 14, 5	LUT 8, 1	24	8	20.79%	20	9.09%	22123	20.79%	8881	16.69%	8.067 $\sigma = 0.01$ µs	2.02 M $\sigma = 0.063$ µs	$\sigma = 0.014$
Fixed 14, 5	LUT 10, 1	24	10	21.02%	20	9.09%	22369	21.02%	9607	18.06%	8.058 $\sigma = 0.111$ µs	2.02 M $\sigma = 0.063$ µs	0.0189 $\sigma = 0.011$

BIBLIOGRAPHY

1. Moore, G. E. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*. - (2006.) - Vol.11. - Nr.3. - 33–35 p. (2006.).
2. Saleh, R. *et al.* System-on-Chip: Reuse and Integration *Proceedings of the IEEE*. - (2006.) - Vol.94. - Nr.6. - 1050–1069 p. (2006.).
3. Nguyen, H. K., Le-Van, T.-V. & Tran, X.-T. A Survey on Reconfigurable System-on-Chips *REV Journal on Electronics and Communications*. - (2018.) - Vol.7. - Nr.3-4. - 74–86 p. (2018.).
4. Gadepally, V. *et al. AI Enabling Technologies: A Survey* - 2019 arXiv: 1905.03592 [cs.AI].
5. Wan, Z. *et al.* A Survey of FPGA-Based Robotic Computing *IEEE Circuits and Systems Magazine*. - (2021.) - Vol.21. - Nr.2. - 48–74 p. (2021.).
6. Hou, N., Yan, X. & He, F A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design *Des Autom Embed Syst*. - (2019.) - Vol.23 - 57–77 p. (2019.).
7. Novickis, R. & Greitāns, M. FPGA Master based on chip communications architecture for Cyclone V SoC running Linux// *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*. ISSN: 2576-3555 - Apr. 2018. - 403–408 p. doi:10.1109/CoDIT.2018.8394842.
8. Dendaluce Jahnke, M., Cosco, F., Novickis, R., Pérez Rastelli, J. & Gomez-Garay, V. Efficient Neural Network Implementations on Parallel Embedded Platforms Applied to Real-Time Torque-Vectoring Optimization Using Predictions for Multi-Motor Electric Vehicles en *Electronics*. - (Feb. 2019.) - Vol.8. - Nr.2. - 250 p. ISSN: 2079-9292 (Feb. 2019.).
9. Setka, V., Jezek, O. & Novickis, R. Modular Signal Processing Unit for Motion Control Applications Based on System-on-Chip with FPGA en// *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. - Zaragoza, Spain: IEEE, Sept. 2019. - 857–863 p. ISBN: 978-1-72810-303-7 doi:10.1109/ETFA.2019.8869121 / URL - https://ieeexplore.ieee.org/document/8869121/ (2020).
10. Novickis, R., Justs, D. J., Ozols, K. & Greitāns, M. An Approach of Feed-Forward Neural Network Throughput-Optimized Implementation in FPGA *Electronics*. - (2020.) - Vol.9. - Nr.12. - 2193 p. (2020.).
11. Novickis, R., Levinskis, A., Kadikis, R., Fescenko, V. & Ozols, K. Functional Architecture for Autonomous Driving and its Implementation *2020 17th Biennial Baltic Electronics Conference (BEC)*. - (2020.) doi:10.1109/bec49624.2020.9276943 / URL - https://ieeexplore.ieee.org/abstract/document/9276943 (2020.).
12. Druml, N. *et al.* Programmable Systems for Intelligence in Automobiles (PRYSTINE) Technical Progress after Year 2 *2020 23rd Euromicro Conference on Digital System Design (DSD)*. - (2020.) doi:10.1109/dsd51259.2020.00065 / URL - https://ieeexplore.ieee.org/document/9217654 (2020.).

13. Druml, N. *et al.* Programmable Systems for Intelligence in Automobiles (PRYSTINE): Final results after Year 3// *2021 24th Euromicro Conference on Digital System Design (DSD)*. - 2021. - 268–277 p. doi:10.1109/DSD53832.2021.00049.

14. Novickis, R. *et al.* Development and experimental validation of high performanceembedded intelligence and fail-operational urban surroundperception solutions of the PRYSTINE project *Applied Sciences*. - (2021.) (2021.).

15. Rihards, N., Vlastimil, S. & Kaspars, O. An exploration of Asynchronous Multi-Processing for Real-Time Control Systems.. en 2020.

16. Rihards, N. & Justs, D. J. Modular architecture suitable for deployment in real-time systems.. en 2020.

17. Rihards, N. Simultaneous access of n-dimensional data in digital circuits.. en 2020.

18. Sieg, W. en// *Philosophy of Mathematics* - 535–630 p. - Elsevier, 2009. ISBN: 978-0-444-51555-1 doi:10.1016/B978-0-444-51555-1.50017-1 / URL - https://linkinghub.elsevier.com/retrieve/pii/B9780444515551500171 (2020).

19. Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem en *Proc London Math Soc*. - (Jan. 1937.) - Vol.s2-42. - Nr.1. Publisher: Oxford Academic - 230–265 p. ISSN: 0024-6115 (Jan. 1937.).

20. Von Neumann, J. *First Draft of a Report on the EDVAC* tech. rep. (United States Army Ordnance Department, Moore School of Electrical Engineering University of Pennsylvania, June 1945).

21. Patt, Y. N. & Patel, S. J. Introduction to computing systems: from bits and Gates to C and beyond. en OCLC: 145555083 ISBN: 978-0-07-124501-2 - Boston: McGraw-Hill Higher Education, 2005.

22. Shen, J. P. & Lipasti, M. H. L. Modern processor design: Fundamentals of superscalar processors. en ISBN: 1-3786-0783-1 978-1-4786-0783-0 - Waveland Press, Inc., 2013.

23. David A. Patterson & John L. Hennessy Computer Organization and Design: the hardware and software interface. ARM edition ISBN: 978-0-12-801733-3 - United States of America: Elsevier, 2017.

24. Kuon, I., Tessier, R. & Rose, J. FPGA Architecture: Survey and Challenges en *Foundations and Trends® in Electronic Design Automation*. - (2007.) - Vol.2. - Nr.2. - 135–253 p. ISSN: 1551-3939, 1551-3947 (2007.).

25. Chu, P. P. RTL hardware design using VHDL. ISBN: 9780471720928 doi:10.1002/0471786411 - John Wiley & Sons, 2006.

26. Nane, R. *et al.* A Survey and Evaluation of FPGA High-Level Synthesis Tools en *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. - (Oct. 2016.) - Vol.35. - Nr.10. - 1591–1604 p. ISSN: 0278-0070, 1937-4151 (Oct. 2016.).

27. NVIDIA corporation CUDA C++ Programming Guide, Design Guide en (2020.) - 404 p. (2020.).

28. Intel corporation Intel FPGA SDK for OpenCL Pro Edition: Programming Guide en (June 2020.) - 231 p. (June 2020.).

29. Demidov, D., Ahnert, K., Rupp, K. & Gottschling, P. Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries *SIAM J. Sci. Comput.*. - (2013.) - Vol.35. - Nr.5. doi:10.1137/120903683 / URL - https://doi.org/10.1137/120903683 (2013.).

30. Liu, L. *et al.* A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications *ACM COMPUTING SURVEYS*. - (2020.) - Vol.52. - Nr.6. ISSN: 0360-0300 doi:{10.1145/3357375} (2020.).

31. Kumar, A., Balakrishnan, M, Jain, M. K. & Gangwar, A. Customizing Embedded Processors for Specific Applications en - 25 p.

32. Liu, D. Embedded DSP Processor Design: Application Specific Instruction Set Processors. ISBN: 9780123741233 doi:10.1016/b978-0-12-374123-3.x5001-5 / URL - https://www.sciencedirect.com/book/9780123741233/embedded-dsp-processor-design - Elsevier, 2008.

33. Galetovic, A. *Intellectual Property and the history of the semiconductor industry* - Universidad de los Andes, May 2018 / URL - https://hooverip2.org/wp-content/uploads/What-Patents-Really-Do-Galetovic-051718.pdf (2020).

34. Flynn, M. J. & Luk, W. Computer system design : system-on-chip. ISBN: 9780470643365 / URL - https://www.wiley.com/en-us/Computer+System+Design\%3A+System+on+Chip-p-9780470643365 - Wiley, 2011.

35. Intel corporation Avalon® Interface Specifications en (May 2020.) - 70 p. (May 2020.).

36. Robert Love Linux Kernel Development - A thorough guide to the design and implementation of the Linux kernel. 3rd Edition ISBN: 978-0-672-32946-3 / URL - https://www.doc-developpement-durable.org/file/Projets-informatiques/cours-&-manuels-informatiques/Linux/Linux%20Kernel%20Development,%203rd%20Edition.pdf (2020) - Pearson Education, 2010.

37. Insight Technologies *State of Linux in the public cloud for enterprises* Solution overview (Red Hat, 2018) / URL - https://www.redhat.com/cms/managed-files/cl-state-of-linux-in-public-cloud-for-enterprises-f11154kc-201802-en_0.pdf (2020).

38. Corbet, J., Rubini, A. & Kroah-Hartman, G. Linux Device Drivers. Third - O'Reilly, Dec. 2010.

39. Michal Nazarewicz A deep dive into CMA en (2012.) / URL - https://lwn.net/Articles/486301/ (2012.).

40. Szeliski, R. Multiple view geometry in computer vision. en OCLC: 171123855 ISBN: 978-0-511-18711-7 978-0-511-18618-9 978-0-511-18895-4 978-0-511-18535-9 978-0-511-18451-2 978-0-511-81168-5 978-1-280-45812-5 / URL - http://dx.doi.org/10.1017/CBO9780511811685 (2019) - 2004.

41. Szeliski, R. Computer Vision: Algorithms and Applications en (2011.) - 979 p. ISSN: 1868-0941 (2011.).

42. Brown, D. Decentering distortion of lenses *Photogramm. Eng.*. - (1966.) (1966.).

43. Fitzgibbon, A. Simultaneous linear estimation of multiple view geometry and lens distortion// *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001.* - Vol.1 - 2001. - I–I p. doi:10.1109/CVPR.2001.990465.

44. Marr, D. Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. en - Nr.2. ISBN: 0-7167-1567 - W. H. Freeman and Company, 1982.

45. Collins, R. A space-sweep approach to true multi-image matching// *Proceedings CVPR IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* - 1996. - 358–363 p. doi:10.1109/CVPR.1996.517097.

46. Scharstein, D., Szeliski, R. & Zabih, R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms// *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001).* - Dec. 2001. - 131–140 p. doi:10.1109/SMBV.2001.988771.

47. Hannah, M. J. Computer Matching of Areas in Stereo Images. PhD thesis (Standford University, 1974.).

48. Kanade, T. & Okutomi, M. A stereo matching algorithm with an adaptive window : theory and experiment en *Image Understanding Workshop*. - (1994.) - 549–557 p. (1994.).

49. Kumar, M. P. & Torr, P. H. S. Improved Moves for Truncated Convex Models en (2011.) - 8 p. (2011.).

50. D. Marr, T. P. Cooperative computation of stereo disparity *Science*. - (1976.) / URL - http://cbcl.mit.edu/people/poggio/journals/marr-poggio-science-1976.pdf (2020) (1976.).

51. Prazdny, K. Detection of binocular disparities en *Biol. Cybern.*. - (June 1985.) - Vol.52. - Nr.2. - 93–99 p. ISSN: 1432-0770 (June 1985.).

52. Scharstein, D. Matching images by comparing their gradient fields en// *Proceedings of 12th International Conference on Pattern Recognition.* - Vol.1 - Jerusalem, Israel: IEEE Comput. Soc. Press, 1994. - 572–575 p. ISBN: 978-0-8186-6265-2 doi:10.1109/ICPR.1994.576363 / URL - http://ieeexplore.ieee.org/document/576363/ (2020).

53. Black, M. J. & Rangarajan, A. On the unification of line processes, outlier rejection, and robust statistics with applications in early vision *International Journal of Computer Vision*. - (1996.) - Vol.19. - Nr.1. - 57–91 p. (1996.).

54. Scharstein, D. & Szeliski, R. Stereo matching with non-linear diffusion// *Proceedings CVPR IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* - 1996. - 343–350 p. doi:10.1109/CVPR.1996.517095.

55. Szeliski, R. & Scharstein, D. Sampling the disparity space image *IEEE Transactions on Pattern Analysis and Machine Intelligence*. - (2004.) - Vol.26. - Nr.3. - 419–425 p. (2004.).

56. Krizhevsky, A., Sutskever, I. & Hinton, G. E. ImageNet classification with deep convolutional neural networks en *Communications of the ACM*. - (May 2017.) - Vol.60. - Nr.6. - 84–90 p. ISSN: 00010782 (May 2017.).

57. Liu, W. *et al.* A survey of deep neural network architectures and their applications en *Neurocomputing*. - (Apr. 2017.) - Vol.234 - 11–26 p. ISSN: 09252312 (Apr. 2017.).

58. Soni, D. *Spiking Neural Networks, the Next Generation of Machine Learning* - Jan. 2018 / URL - https://towardsdatascience.com/spiking-neural-networks-the-next-generation-of-machine-learning-84e167f4eb2b (2018).

59. Qiao, L., Zhao, H., Huang, X., Li, K. & Chen, E. A Structure-Enriched Neural Network for network embedding English *Expert Syst. Appl.*. - (Mar. 2019.) - Vol.117 WOS:000449892000021 - 300–311 p. ISSN: 0957-4174 (Mar. 2019.).

60. Ince, T., Kiranyaz, S., Eren, L., Askar, M. & Gabbouj, M. Real-Time Motor Fault Detection by 1-D Convolutional Neural Networks English *IEEE Trans. Ind. Electron.*. - (Nov. 2016.) - Vol.63. - Nr.11. WOS:000388622100042 - 7067–7075 p. ISSN: 0278-0046 (Nov. 2016.).

61. Kadikis, R. Recurrent neural network based virtual detection line// *Tenth International Conference on Machine Vision (ICMV 2017)*. - Vol.10696 - International Society for Optics and Photonics, Apr. 2018. - 106961V p. doi:10.1117/12.2309772 / URL - https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10696/106961V/Recurrent-neural-network-based-virtual-detection-line/10.1117/12.2309772.short (2018).

62. Wu, Y. & Lee, T. Reducing Model Complexity for DNN Based Large-Scale Audio Classification. English WOS:000446384600066 ISBN: 978-1-5386-4658-8 - New York: Ieee, 2018.

63. Haykin, S. Neural Networks: A Comprehensive Foundation. 2nd ISBN: 978-0-13-273350-2 - Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

64. Fine, T. L. Feedforward Neural Network Methodology. 1st ISBN: 978-0-387-98745-3 - Cornell University, Ithaca, NY, USA: Springer-Verlag New York, 1999.

65. Chakradhar, S., Sankaradas, M., Jakkula, V. & Cadambi, S. A dynamically configurable coprocessor for convolutional neural networks// *ACM SIGARCH Computer Architecture News*. - Vol.38 - ACM, 2010. - 247–257 p. / URL - http://dl.acm.org/citation.cfm?id=1815993 (2017).

66. Suda, N. *et al.* Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks en// *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate*. - ACM Press, 2016. - 16–25 p. ISBN: 978-1-4503-3856-1 doi:10.1145/2847263.2847276 / URL - http://dl.acm.org/citation.cfm?doid=2847263.2847276 (2017).

67. Zhang, C. *et al.* Optimizing fpga-based accelerator design for deep convolutional neural networks// *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. - ACM, 2015. - 161–170 p. / URL - http://dl.acm.org/citation.cfm?id=2689060 (2017).

68. Foumani, S. N. A. An FPGA Accelerated Method for Training Feed-forward Neural Networks Using Alternating Direction Method of Multipliers and LSMR. MA thesis (Imperial College London, Department of Computing, 2020.).

69. Blott, M. *et al.* FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks *ACM Transactions on Reconfigurable Technology and Systems*. - (2018.) doi:10.1145/3242897 (2018.).

70. Guan, Y. *et al.* FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates// *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ISSN: 978-1-5386-4038-8 - 2017. doi:10.1109/FCCM.2017.25.

71. Hariprasath, S. & Prabakar, T. N. FPGA implementation of multilayer feed forward neural network architecture using VHDL// *Computing, Communication and Applications (ICCCA), 2012 International Conference on.* - IEEE, 2012. - 1–6 p. / URL - http://ieeexplore.ieee.org/abstract/document/6179225/ (2017).

72. Youssef, A., Mohammed, K. & Nasar, A. A Reconfigurable, Generic and Programmable Feed Forward Neural Network Implementation in FPGA// *2012 UKSim 14th International Conference on Computer Modelling and Simulation.* - IEEE, 2012. - 9–13 p. ISBN: 978-1-4673-1366-7 978-0-7695-4682-7 doi:10.1109/UKSim.2012.12 / URL - http://ieeexplore.ieee.org/document/6205543/ (2017).

73. Zamanlooy, B. & Mirhassani, M. Efficient VLSI Implementation of Neural Networks With Hyperbolic Tangent Activation Function *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. - (2014.) - Vol.22. - Nr.1. - 39–48 p. ISSN: 1063-8210 (2014.).

74. Yuan Jing, Youssefi, B., Mirhassani, M. & Muscedere, R. An efficient FPGA implementation of Optical Character Recognition for License Plate Recognition en// *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE).* - IEEE, 2017. - 1–4 p. ISBN: 978-1-5090-5538-8 doi:10.1109/CCECE.2017.7946734 / URL - http://ieeexplore.ieee.org/document/7946734/ (2018).

75. Oliveira, J. G. M., Moreno, R. L., de Oliveira Dutra, O. & Pimenta, T. C. Implementation of a reconfigurable neural network in FPGA en// *2017 International Caribbean Conference on Devices, Circuits and Systems (ICCDCS).* - Cozumel, Mexico: IEEE, 2017. - 41–44 p. ISBN: 978-1-5386-1962-9 doi:10.1109/ICCDCS.2017.7959699 / URL - http://ieeexplore.ieee.org/document/7959699/ (2018).

76. Hajduk, Z. Reconfigurable FPGA implementation of neural networks en *Neurocomputing*. - (2018.) - Vol.308 - 227–234 p. ISSN: 09252312 (2018.).

77. Khan, A., Sohail, A., Zahoora, U. & Qureshi, A. S. A survey of the recent architectures of deep convolutional neural networks en *Artif Intell Rev*. - (2020.) - Vol.53. - Nr.8. - 5455–5516 p. ISSN: 1573-7462 (2020.).

78. Shalf, J. & Leland, R. Computing beyond Moore's Law *Computer*. - (2015.) - Vol.48 - 14–23 p. (2015.).

79. *nRF52832 Product Specification* v1.4 Nordic Semiconductors - Oct. 2017 553 / URL - https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.4.pdf.

80. *HSA Platform System Architecture Specification* http://www.hsafoundation.com/ HSA foundation - Jan. 2016.

81. Liu, W., Chen, H. & Ma, L. Moving object detection and tracking based on ZYNQ FPGA and ARM SOC// *IET International Radar Conference 2015.* - Oct. 2015. - 1–4 p.

82. EETimes *EETimes - Why choose Linux for embedded development projects?* - Library Catalog: www.eetimes.com Section: Report - Dec. 2002 / URL - https://www.eetimes.com/why-choose-linux-for-embedded-development-projects/ (2020).

83. Bailey, D. G. The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing// *Proceedings of the 9th International Conference on Distributed Smart Cameras.* - ACM New York, NY, USA, Sept. 2015. - 134–139 p.

84. Cortes, I., Velez, I. & Irizar, A. High level synthesis using Vivado HLS for Zynq SoC: Image processing case studies// *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*. - Nov. 2016.

85. Sadri, M., Weis, C., Wehn, N. & Benini, L. Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ// *FPGAworld '13 Proceedings of the 10th FPGAworld Conference*. - 2013.

86. Molanes, R. F., Salgado, F., Fariña, J. & Rodríguez-Andina, J. J. Characterization of FPGA-master ARM communication delays in Cyclone V devices// *41st Annual Conference of the IEEE Industrial Electronics Society*. - 2015. - 4229–4234 p.

87. Vogel, P., Marongiu, A. & Benini, L. An Evaluation of Memory Sharing Performance for Heterogeneous Embedded SoCs with Many-Core Accelerators// *COSMIC '15 Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*. - 2015.

88. Altera corp. *Cyclone V Hard Processor System Technical Reference Manual* - Oct. 2016.

89. *Embedded Peripherals IP User Guide* Intel corp. - Dec. 2016.

90. *Avalon Interface Specification* Altera corp. - Dec. 2015.

91. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite en (2003.) - 306 p. (2003.).

92. Intel corporation Qsys Interconnect en (Aug. 2014.) - 65 p. (Aug. 2014.).

93. Corbet, J., Rubini, A. & Kroah-Hartman, G. Linux Device Drivers. Third - O'Reilly, Dec. 2010.

94. Nazarewicz, M. A deep dive into CMA (Mar. 2012.) https://www.lwn.net/ (Mar. 2012.).

95. Group, E. T. EtherCAT Technology Group | EtherCAT / URL - https://www.ethercat.org/en/technology.html.

96. Jasperneite, J., Schumacher, M. & Weber, K. Limits of increasing the performance of Industrial Ethernet protocols// *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. - 2007. - 17–24 p. doi:10.1109/EFTA.2007.4416748.

97. Šetka, V., Ježek, O. & Novickis, R. Modular Signal Processing Unit for Motion Control Applications Based on System-on-Chip with FPGA// *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. - 2019. - 857–863 p. doi:10.1109/ETFA.2019.8869121.

98. Sangmin Chon *What it Takes to do Efficient and Cost-Effective Real-Time Control with a Single Microcontroller The C2000™Advantage* White paper (Texas Instruments, 2011) / URL - https://www.ti.com/lit/wp/spry157/spry157.pdf?ts=1637148859226\&ref_url=https\%253A\%252F\%252Fwww.google.com\%252F (2021).

99. Walls, C. Multicore basics: AMP and SMP - Embedded.com / URL - https://www.embedded.com/multicore-basics-amp-and-smp.

100. Kim, B. & Choi, M. Design and Analysis of Multiple OS Implementation on a Single ARM-Based Embedded Platform *Sustainability*. - (2017.) - Vol.9. - Nr.5. - 684 p. (2017.).

101. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition en (2000.) - 2736 p. (2000.).

102. Kerrisk, M. *Linux Programmer's Manual. sysfs - a filesystem for exporting kernel objects* Linux - Mar. 2021 / URL - https://man7.org/linux/man-pages/man5/sysfs.5.html.

103. F. Buschmann, K. Henney & D. C. Schmidt Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. - Chichester: Wiley, 2007.

104. NVIDIA Corporation NVIDIA GPU Programming Guide en (2005.) - 80 p. (2005.).

105. Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System* version ROS Melodic Morenia - / URL - https://www.ros.org.

106. Maruyama, Y., Kato, S. & Azumi, T. Exploring the performance of ROS2// *2016 International Conference on Embedded Software (EMSOFT)*. - Oct. 2016. - 1–10 p. doi:10.1145/2968478.2968502.

107. Maruyama, Y., Kato, S. & Azumi, T. Exploring the performance of ROS2// *2016 International Conference on Embedded Software (EMSOFT)*. - 2016. - 1–10 p. doi:10.1145/2968478.2968502.

108. Akgul, F. ZeroMQ. ISBN: 978-1-78216-104-2 - Packt Publishing, 2013.

109. Fu, Y. *et al.* Deep Learning with INT8 Optimization on Xilinx Devices en (2017.) - 11 p. (2017.).

110. Dettmers, T. 8-Bit Approximations for Parallelism in Deep Learning en *arXiv:1511.04561 [cs]*. - (Nov. 2015.) arXiv: 1511.04561 / URL - http://arxiv.org/abs/1511.04561 (2018) (Nov. 2015.).

111. AXI DMA v7.1, LogiCORE IP Product Guide (Apr. 2018.) - 95 p. (Apr. 2018.).

112. Zynq-7000 All Programmable SoC Technical Reference Manual (UG585) en (2016.) - 1863 p. (2016.).

113. Galceran-Oms, M., Cortadella, J. & Kishinevsky, M. Automatic Pipelining of Elastic Systems. en PhD thesis (UNIVERSITAT POLITÈCNICA DE CATALUNYA DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS, June 2011.) - 201 p.

114. Bayer, B. E. *US Patent* - Nr.US3971065A. (1976).

115. Li, X., Gunturk, B. & Zhang, L. Image demosaicing: a systematic survey en// (eds Pearlman, W. A., Woods, J. W. & Lu, L.) - San Jose, CA, Jan. 2008. - 68221J p. doi:10.1117/12.766768 / URL - http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.766768 (2020).

116. Yu, H. & Leeser, M. Automatic Sliding Window Operation Optimization for FPGA-Based// *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. - 2006. - 76–88 p. doi:10.1109/FCCM.2006.29.

117. Hagara, M., Stojanović, R., Bagala, T., Kubinec, P. & Ondráček, O. Grayscale image formats for edge detection and for its FPGA implementation *Microprocessors and Microsystems*. - (2020.) - Vol.75 - 103056 p. ISSN: 0141-9331 (2020.).

118. Huntsberger, T. & Descalzi, M. Color edge detection *Pattern Recognition Letters*. - (1985.) - Vol.3. - Nr.3. - 205–209 p. ISSN: 0167-8655 (1985.).

119. Cook, J. D. Three algorithms for converting to grayscale / URL - https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/.

120. Zemčík, P., Přibyl, B., Herout, A. & Seeman, M. Accelerated image resampling for geometry correction *Journal of Real-Time Image Processing*. - (2011.) - Vol.8. - Nr.4. - 369–377 p. (2011.).

121. Clapa, J., Blasinski, H., Grabowski, K. & Sekalski, P. A fisheye distortion correction algorithm optimized for hardware implementations *2014 Proceedings of the 21st International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)*. - (2014.) doi:10.1109/mixdes.2014.6872232 / URL - https://ieeexplore.ieee.org/document/6872232 (2014.).

122. Andraka, R. A survey of CORDIC algorithms for FPGA based computers *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. - (1998.) doi:10.1145/275107.275139 / URL - https://dl.acm.org/doi/10.1145/275107.275139 (1998.).

123. Guo, W., Zhang, C. & Zheng, Y. R. Real-Time Image Distortion Correction System Based on Improved Bilinear Interpolation Algorithm *Applied Mechanics and Materials*. - (2014.) - Vol.513-517 - 3773–3776 p. (2014.).

124. Aho, E., Vanne, J., Hämäläinen, T. D. & Kuusilinna, K. Configurable Implementation of Parallel Memory Based Real-Time Video Downscaler *Microprocess. Microsyst.*. - (Aug. 2007.) - Vol.31. - Nr.5. - 283–292 p. ISSN: 0141-9331 (Aug. 2007.).

125. Junger, C., Hess, A., Rosenberger, M. & Notni, G. FPGA-based lens undistortion and image rectification for stereo vision applications *Photonics and Education in Measurement Science 2019*. - (2019.) (eds Zagar, B., Rosenberger, M. & Dittrich, P.-G.) doi:10.1117/12.2530692 / URL - https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11144/1114416/FPGA-based-lens-undistortion-and-image-rectification-for-stereo-vision/10.1117/12.2530692.full (2019.).

126. Soderquist, P. & Leeser, M. Division and square root: choosing the right implementation *IEEE Micro*. - (1997.) - Vol.17. - Nr.4. - 56–66 p. (1997.).

127. Oberman, S. Floating point division and square root algorithms and implementation in the AMD-K7/sup TM/ microprocessor// *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*. - 1999. - 106–115 p. doi:10.1109/ARITH.1999.762835.

128. Chiew, W. M., Lin, F. & Soon, S. H. A Novel Embedded Interpolation Algorithm with Negative Squared Distance for Real-Time Endomicroscopy *ACM Transactions on Embedded Computing Systems (TECS)*. - (2016.) - Vol.15 - 1 –19 p. (2016.).

129. *Why is fpga-gpu heterogeneity the best option for embedded deep neural networks?* Preprint - ResearchGate, 2021 / URL - https://www.researchgate.net/publication/348980299_Why_is_FPGA-GPU_Heterogeneity_the_Best_Option_for_Embedded_Deep_Neural_Networks.

130. Zabih, R. & Woodfill, J. Non-parametric local transforms for computing visual correspondence// *Computer Vision — ECCV '94*. (ed Eklundh, J.-O.) - Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. - 151–158 p. ISBN: 978-3-540-48400-4.

131. *i.MX 6Dual/6Quad Applications Processors for Consumer Products* Rev 6. NXP - Nov. 2018 171 / URL - https://www.nxp.com/docs/en/reference-manual/i.MX_Reference_Manual_Linux.pdf.

132. Bont, F. d. *Zynq UltraScale+ MPSoC for the System Architect* nl-NL Library Catalog: www.core-vision.nl - / URL - https://www.core-vision.nl/events/zynq-ultrascale-mpsoc-for-the-system-architect-11/ (2020).