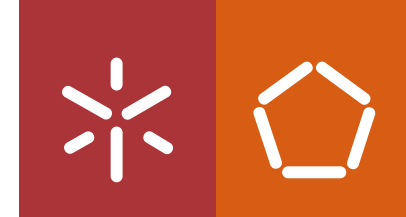


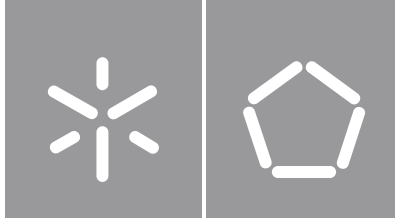


Acceleration on FPGA of an SVM Classifier for Road Condition Sensor

Marcelo Quintela Alves

Universidade do Minho
Escola de Engenharia





Universidade do Minho

Escola de Engenharia

Marcelo Quintela Alves

Acceleration on FPGA of an SVM Classifier for Road Condition Sensor

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e
Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos. Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada. Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



Atribuição-CompartilhaIgual

CC BY-SA

<https://creativecommons.org/licenses/by-sa/4.0/>

Acknowledgements

I would like to preface this thesis by thanking everyone who helped, inspired and supported me on this journey. I would first like to thank Dr. Jorge Cabral for inviting me to take part in such an interesting project and for mentoring me during my thesis, but mostly for his kindness and continuous belief in me. My sincere thanks to João Carvalho, as both this work and myself have benefited greatly from his immense knowledge, his trust and his guidance. I also want to express my deep gratitude to all my lab friends, who helped me in countless ways in the time we spent together and taught me that science requires devotion, discipline and curiosity.

Special thanks to my two loving parents and my cool sister. I am forever in your debt, for all the support, love and patience you've shown me. I am incredibly lucky to have you, and I can't thank you enough for all your support. To my girlfriend, my dear Elena, thank you ever so much. Thank you for believing in me, and for believing in us especially when circumstances kept us apart for much too long. I love you, silly goose.

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project n° 037902; Funding Reference: POCI-01-0247-FEDER-037902].

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Aceleração em FPGA de um classificador SVM para o Road Condition Sensor

O conceito de condução autónoma captou a imaginação de engenheiros, escritores de ficção científica e também condutores presos no trânsito, desde o surgimento do automóvel moderno no início do século 20. A promessa de um veículo que se encarregue do volante e dos pedais, capaz de segurança e conforto total pode parecer uma fantasia improvável - no entanto, a indústria prepara-se para a tornar numa realidade nas próximas décadas.

A transição para a condução inteiramente autónoma apresenta graves obstáculos. A próxima geração de veículos autónomos requer os algoritmos de inteligência artificial mais avançados, que devem ser integrados de uma forma segura e não dispendiosa. Estas novas tecnologias não são compatíveis com os sistemas empregues nos carros de hoje. O futuro do automobilismo depende da adoção de novas plataformas capazes de metamorfose.

Esta tese de mestrado apresenta a adaptação de um protótipo do sensor de condição do piso para uma arquitetura baseada em FPGAs. O aparelho original, concebido para ser montado na dianteira do carro, é munido de sensores óticos sofisticados e integra algoritmos de classificação pré-treinados. Esta configuração será capaz de classificar a estrada por onde o veículo se desloca com base na presença (ou ausência) de água, gelo ou neve.

Os artefactos desenvolvidos neste trabalho demonstram: 1) a implementação em hardware reconfigurável de um sistema de aquisição de dados de alta performance, 2) uma arquitetura para algoritmos SVM otimizada para FPGAs, e 3) a integração do sistema de controlo de temperatura num único circuito integrado. A validade da abordagem proposta é demonstrada, enquanto que as maiores vantagens e desvantagens são discutidas no contexto de um projeto que dispõem de tempo e recursos monetários limitados.

Palavras-chave: Aceleração em hardware, FPGA, Sistemas embebidos, SVM.

Abstract

Acceleration on FGPA of an SVM Classifier for Road Condition Sensor

Ever since the advent of mass-produced affordable car in the early 20th century, the concept of autonomous driving has captivated the minds of engineers, science-fiction novelists and exhausted commuters alike. The promise of a vehicle that takes control of the wheel and pedals, while also providing absolute safety, comfort and precision may seem unlikely - and yet, many project it to become a reality within our lifetimes.

Transitioning to fully autonomous driving poses a grave set of challenges. The next generation of vehicles mandates the use of near bleeding-edge data acquisition and processing systems, while still preserving reliability and cost-effectiveness. These modern algorithms and data models require more processing power, and are in a constant process of evolution incompatible with the current processing architectures. For the future of mobility to materialize, it is imperative a shift to platforms capable of metamorphosis.

This thesis covers the adaptation of an existing road condition sensor prototype for an FPGA platform. The apparatus, intended to be mounted under a vehicles license plate, exploits a refined optical sensing solution mated to pre-trained AI-inference algorithms. These produce a classification of the driving surface, categorized in terms of the presence (or absence) of water, in either liquid or solid form.

Some of the artefacts produced for this work include: 1) a both simple and high-performing data acquisition system implemented in reconfigurable logic, 2) a scalable, fixed-point linear SVM architecture optimized for the FPGA's architecture, and 3) the integration of a thermal management loop, all in a single chip. The validity of this approach is demonstrated, while the main advantages and drawbacks are discussed in the context of a resource and time constrained research and innovation project.

Key Words: Embedded systems, FPGA, Hardware acceleration, SVM.

Contents

1	Introduction	1
1.1	Background	2
1.2	Motivation	4
1.3	Problem Statement	5
1.4	Objectives	7
1.5	State-of-the-Art	7
1.6	Thesis Structure	9
2	Research Platform and Tools	10
2.1	Platform Requirements	10
2.2	Hardware Specification	11
2.2.1	Zynq-7000 SoC Family	11
2.2.2	Block RAM	15
2.2.3	DSP48E1 Slice	16
2.2.4	ZYBO Z7-10	17
2.2.5	Sensors	18
2.2.5.1	Analog-to-Digital Converter	19
2.2.5.2	Temperature Sensors	20
2.2.6	Actuators	21
2.3	Software Specification	21
3	Lasers and Photodetector Controller	23
3.1	Analysis	23
3.2	Design	24
3.2.1	SPI Module	25
3.2.2	ADS8332 Controller Module	29
3.2.3	Laser Controller Module	34

3.2.4	Test Cases	34
3.3	Implementation & Evaluation	36
3.3.1	SPI Module	36
3.3.2	ADS8332 Controller Module	41
3.3.3	Laser Controller Module	43
3.4	Discussion	44
4	LASER2SVM Module	46
4.1	Analysis	46
4.2	Design	47
4.3	Implementation & Evaluation	51
4.4	Discussion	55
5	Support Vector Machine	57
5.1	Analysis	57
5.2	Design	63
5.2.1	Test Cases	67
5.3	Implementation & Evaluation	69
5.3.1	Test Results	73
5.3.2	RCS2FPGA Architecture	75
5.4	Discussion	80
6	Conclusion and Future Work	81
6.1	Future Work	82

List of Figures

1.1	Block Diagram of the original prototype of the Road Condition Sensor.	6
2.1	Zynq-7000 SoC Architectural Overview	12
2.2	Detail of the Zynq-7000's column-based PL layout.	13
2.3	PL Interface to PS Memory Subsystem.	14
2.4	Multiple supported Zynq-7000 BRAM configurations.	15
2.5	Basic DSP48E1 Slice Functionality.	16
2.6	ZYBO Z7-10 Board.	17
2.7	ADS8332 Functional Block Diagram	19
2.8	ADT7301 Functional Block Diagram	20
3.1	SPI bus architecture of the previous RCS prototype.	26
3.2	ADS8332 serial interface timing diagram.	27
3.3	SPI Module Functional Block Diagram.	28
3.4	SPI Module finite-state machine diagram.	30
3.5	ADS8332 Controller Module finite-state machine diagram.	33
3.6	Laser Controller Module finite-state machine diagram.	35
3.7	Block Design View of SPI Module AXI test.	38
3.8	SPI Module's test case results.	39
3.9	Detail of ADS8332 Controller's initialization process	42
4.1	Illustration of selected CDC Handshake mechanism.	51
4.2	Divider Module's simulation results	56
5.1	SVM hyperplane.	58
5.2	SVM Module's functional block diagram.	65
5.3	Detail of systolic array submodule.	66
5.4	Detail of Multiplier–Accumulator (MAC) unit.	67
5.5	Block diagram of SVM Module's implementation.	70

5.6	Block Design View of SVM Module AXI test.	73
5.7	Functional Block Diagram of RCS2FPGA architecture.	75
5.8	“Device View” of SVM Module’s implementation.	78
5.9	“Device View” of RCS2FPGA system implementation.	79

List of Tables

1.1	Summary of levels of diving automation, adapted from SAE J3016_201806.	3
3.1	Summary of ADS8332 serial interface timing requirements.	27
3.2	Summary of ADS8332 Command Set.	31
3.3	Input/output connections between ZYBO Z7-10 and Arduino Uno REV3	36
3.4	SPI Module resource utilization (WORD_SIZE=16).	40
3.5	Summary of ADT7301 serial interface timing requirements.	40
3.6	Summary of DAC8162 serial interface timing requirements.	41
3.7	ADS8332 Controller Module resource utilization.	43
3.8	Laser Controller Module resource utilization.	44
4.1	LASER2SVM Module resource utilization.	53
5.1	Distribution of values of support vector matrix.	64
5.2	Analysis of kernel coefficients required for each class confrontation.	67
5.3	Analysis of BRAM contents in the final SVM Module's implementation.	71
5.4	SVM Module resource utilization.	72
5.5	RCS2FPGA Architecture's resource utilization.	76

List of Listings

1	SPI Module's VHDL interface.	37
2	Detail of the SPI Module's C++ testing application deployed on the ZYBO Z7-10's PS. . .	38
3	ADS8332 Controller Module's VHDL interface.	42
4	Laser Controller Module's VHDL interface.	44
5	Non-restoring division's main computation loop.	49
6	Example of non-recurring division of -12 by 15, with P=8.	50
7	LASER2SVM Module's VHDL interface.	52
8	Non-restoring Division Module's VHDL interface.	53
9	LASER2SVM Module's CDC Handshake implementation.	54
10	Constant definitions of SVM algorithm.	60
11	Detail of Kernel Computation stage of the SVM algorithm.	61
12	Detail of "one-vs-one" multi-class approach of the SVM algorithm.	62
13	Detail of voting procedure of the SVM algorithm.	62
14	Kernel value saturation, to ensure arithmetic equivalence between the original and the implemented SVM algorithm.	68
15	SVM Module's VHDL package interface.	69
16	SVM Module's top module VHDL interface.	70
17	Detail of the SVM Module's C++ testing application deployed on the ZYBO Z7-10's PS. .	74

List of Abbreviations

SoC	System-on-Chip.
PL	Programmable Logic.
PS	Processing System.
BRAM	Block RAM.
RAM	Random-Access Memory.
SPI	Serial Peripheral Interface.
AXI	AMBA Advanced eXtensible Interface.
DDT	Dynamic Driving Task.
ODD	Operational Design Domain.
OEDR	Object and Event Detection and Response.
ADS	Automated Driving System.
RCS	Road Condition Sensor.
PCB	Printed Circuit Board.
ADC	Analog-to-digital Converter.
CPU	Central Processing Unit.
CAN	Controller Area Network.
FPGA	Field-programmable Gate Array.
APU	Application Processing Unit.
SIMD	Single Instruction, Multiple Data.
VFP	Vector Floating Point.
DSP	Digital Signal Processing.
SCU	Snoop Control Unit.
SRAM	Static Random-Access Memory.
DMA	Direct Memory Access.
DDR	Double Data Rate.
SDRAM	Synchronous dDynamic Random-Access Memory.

GPIO	General-Purpose Input/Output.
UART	Universal Asynchronous Receiver-Transmitter.
I ² C	Inter-Integrated Circuit.
CLB	Configurable Logic Blocks.
XADC	Xilinx Analog-to-Digital Converter.
AMS	Analog Mixed Signal.
PCI	Peripheral Component Interconnect.
AMBA	Advanced Microcontroller Bus Architecture.
ROM	Read-Only Memory.
FIFO	First In, First Out.
ECC	Error Correction Code.
LUT	Lookup Table.
USB	Universal Serial Bus.
LED	Light-Emitting Diode.
NTC	Negative Temperature Coefficient.
SPS	Samples Per Second.
HDL	Hardware Description Language.
DUT	Device Under Testing.
HLS	High-Level Synthesis.
RTL	Register-Transfer Level.
IP	Intellectual Property.
μC	Microcontroller.
FSM	Finite-State machine.
SCLK	Serial clock.
CPOL	Clock polarity.
CPHA	Clock phase.
CS	Chip Select.
MOSI	Master Output, Slave Input.
MISO	Master Input, Slave Output.
CCLK	Conversion clock.
PISO	Parallel In, Serial Out.
SIPO	Serial In, Parallel Out.

MSB	Most Significant Bit.
ISR	Interrupt Service Routine.
SLD	Semiconductor Laser Diodes.
PID	Proportional–Integral–Derivative.

Chapter 1

Introduction

“Computers were built in the late 1940s because mathematicians like John von Neumann thought that if you had a computer—a machine to handle a lot of variables simultaneously—you would be able to predict the weather (...) They believed that prediction was just a function of keeping track of things. If you knew enough, you could predict anything. That’s been a cherished scientific belief since Newton.”

“And?”

“Chaos theory throws it right out the window.”

—Michael Crichton, *Jurassic Park* (1990)

The main challenge for the automotive sector nowadays is to reach the maximum level of autonomous driving, in which no driver intervention is required at all and even the inclusion of a steering wheel is purely optional. While some high-end modern vehicles already exhibit some form of autonomy, the technology is nowhere near of becoming ubiquitous or reaching a feature-complete, mature form. When fully realized, autonomous driving will enable new levels of safety and reliability to everyday personal transportation. Some experts even predict profound disruptive effects onto other industries, such as insurance providers, urban planning, housing and healthcare. As it always is, change in the automotive industry is driven by technological breakthroughs, with one of the most crucial being the ability to embed artificial intelligence directly on safety-critical, low-latency sensors.

This Chapter provides a descriptive prelude to this thesis. It begins by discussing the motivation behind this work, and contextualizing the thesis within the overarching research project “Sensible Car”. It includes a brief review of an earlier prototype, developed for the aforementioned project, that is of particular relevance to this thesis. This is followed by a clear establishment of the problem statement and the extent to which the research area will be explored. The target research questions of this thesis

are listed, along with the methodology adopted to answer them. Then, a survey on related work and publications that address, in some capacity, the topic of performing SVM inference on FPGAs is provided. Lastly, the Chapter organization of this thesis is described. This Chapter is organized as follows: Section 1.1 provides an introductory definition of autonomous vehicles; Section 1.2 explains the motivation behind this thesis; Section 1.3 defines the problem statement, followed by a description of the scope of this work in Section 1.4; Section 1.5 provides a review of the state-of-the-art in FPGA implementations of SVM inference; to conclude, Section 1.6 presents the structure of this document.

1.1 Background

The degree of automation found in cars has steadily increased over the past few decades, with systems like anti-lock braking systems, air-bags and automatic transmissions becoming common-place features in the world's car park. The concept of "autonomous vehicle" has been used to describe a wide spectrum of existing and non-existing vehicles, compromising communication and collaboration efforts. The need for common terminology was answered in 2014 by SAE International, an automotive standardization body that published the J3016 standard. This standard introduces a classification system, along with supporting definitions, that has been widely adopted across multiple disciplines [1]. It identifies six levels of driving automation ranging from no automation to full autonomy, summarized below in Table 1.1. This Section provides a summary of this standard (revised version, 15th June 2018), detailing the current state-of-affairs regarding the commercial deployment of driving automation systems.

The vast majority of all road vehicles ever made completely lack self-driving capabilities, and thus earn the "Level 0" classification. The driver is designated as the sole provider of the "dynamic driving task", which the standard defines as "all the real-time operational and tactile functions required to operate a vehicle in on-road traffic". This activity can be broken down into several, which include steering, braking and accelerating. The lowest tier of automation is "Level 1", where the vehicle possesses mechanisms which aid in either steering or accelerating/decelerating, but not both simultaneously, as the driver is expected to perform the remaining activity — some examples include Cruise control [2] and Lane Keeping Assist [3]. "Level 2" systems are able to control the vehicle's speed and direction simultaneously, but still require that the driver remain fully attentive to the driving environment. The third level, "Conditional Driving Automation", can deal with situations that require an immediate response, allowing the driver to divert his attention from the driving task. The system will still prompt the driver when it encounters situations it cannot resolve, such as navigating road construction sites or obeying traffic police instructions. Car manufacturers

Level	Name	Narrative definition	DDT		DDT Fallback	ODD
			Sustained lateral and longitudinal vehicle motion control	OEDR		
Driver performs part or all of the DDT						
0	No Driving Automation	The performance by the driver of the entire DDT, even when enhanced by active safety systems.	Driver	Driver	Driver	n/a
1	Driver Assistance	The sustained and ODD-specific execution by a driving automation system of either the lateral or the longitudinal vehicle motion control subtask of the DDT (but not both simultaneously) with the expectation that the driver performs the remainder of the DDT.	Driver and System	Driver	Driver	Limited
2	Partial Driving Automation	The sustained and ODD-specific execution by a driving automation system of both the lateral and longitudinal vehicle motion control subtasks of the DDT with the expectation that the driver completes the OEDR subtask and supervises the driving automation system.	System	Driver	Driver	Limited
ADS (“System”) performs the entire DDT (while engaged)						
3	Conditional Driving Automation	The sustained and ODD-specific performance by an ADS of the entire DDT with the expectation that the DDT fallback-ready user is receptive to ADS-issued requests to intervene, as well as to DDT performance-relevant system failures in other vehicle systems, and will respond appropriately.	System	System	Fallback-ready user (becomes the driver during fallback)	Limited
4	High Driving Automation	The sustained and ODD-specific performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will respond to a request to intervene.	System	System	System	Limited
5	Full Driving Automation	The sustained and unconditional (i.e., not ODD-specific) performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will respond to a request to intervene.	System	System	System	Unlimited

Table 1.1: Summary of levels of driving automation, adapted from SAE J3016_201806.

will only accept liability for accidents where their system is fully in charge, which isn't always clear in "Level 3" systems — consequently, many have concentrated their research efforts in higher automation systems which completely forego driver intervention. The fourth SAE level, "High Driving Automation", describes an ADS that can take complete control of the driving task, to a point where the human becomes a mere passenger. However, he still has the option to manually override and take control over the vehicle if he so desires — indicating the vehicle retains a conventional cockpit, with steering wheel and pedals. The SAE J3016 standard provides several examples of "Level 4" where it highlights their limitations: the system may only be engaged within a limited urban area where its operating speed is also constrained. These limitations differentiate it from a "Full Driving Automation" system, where driving automation is sustained, unconditional and said to be fully realized. The vehicle can go anywhere and perform any task that an experienced human driver would be able to perform, regardless of the starting and end points or intervening road, traffic, and weather conditions. A conventional cockpit can be completely foregone, since there is no longer the need for a human driver — in this case, the standard defines the human's role as "passenger" or "driverless operation dispatcher" while the ADS is engaged.

As of writing this thesis, the highest level of driving automation commercially available is "Level 2", found mostly in luxury/premium vehicles. Recently, Audi's intentions of commercially debuting a "Level 3" Automated Driving System were foiled due to liability issues and the lack of a legal framework [4], highlighting the challenges that come with commercializing the software and hardware that enable highly automated and intelligent systems.

1.2 Motivation

As the following paragraphs highlight, the collection of companies working to produce self-driving cars is ample and diverse. The aim of this Section is not to produce an exhaustive list, but to provide a profile of the main contributors and to showcase how this technology has captured the interest of both well-established car manufacturers, world-class leading technology multinational corporations, start-ups and academia researchers alike from all over the world.

The author denotes that much of the development and testing activity on autonomous driving technology directly involves car manufacturers, either by initiative or collaboration with technological partners. Many have made strategic investments in specialized start-up companies, such as General Motors' \$1 billion acquisition of Cruise [5] or Ford's \$1 billion and later Volkswagen AG's \$2.6 billion investment in Argo AI [6] [7]. Recently, some manufacturers have commercially debuted partially automated driving

systems (classified as “Level 2” on the SAE J3016 spectrum [1]), such as Tesla’s Autopilot [6]. Other carmakers that have delved into researching and developing these systems include Audi, BMW, Daimler, FCA, Toyota, Honda, Nissan, Kia and Hyundai.

Artificial intelligence and data analytics are some of the key resources that must be explored to achieve fully autonomous systems. This is highlighted by several dominant companies in the information industry who have made continuous investments in self-driving technology. Some examples include Waymo, a spin-off of Google’s self-driving car project known for their extensive road testing of autonomous vehicles [8] [9], and Baidu (the search engine market leader in China at the time of writing) which began research and development on autonomous cars in 2013 and has since made substantial investments in the technology [10] [11]. Nvidia has, throughout the years, devised high performance AI solutions for the automotive sector [12] [13] and established partnerships spanning multiple industries [14] [15]. Mobileye is an Israeli company well known for its extensive history in the development of driverless cars [16] [17] – it was purchased by Intel in 2017, one of the world’s leading companies in semiconductor technology, for \$15.3 billion [18].

Ride sharing companies have also invested heavily in autonomous car technology, eager to adapting the technology for their services: Uber’s subsidiary Uber ATG [19] does research to allow for driverless taxi and delivery services, while Lyft [5] has partnered with car manufacturers and other tech companies [20] [21] [22].

The pursuit of fully autonomous driving has also led to substantial funding of academic research projects. Toyota has established joint research centres with Stanford University [23] and the Massachusetts Institute of Technology [24]. The University of Oxford and the University of Cambridge both took part in a multi-year, government- and industry-funded project for the introduction of self-driving vehicles into the UK, which concluded in 2018 [25]. The Carnegie Mellon University has conducted research on self-driving cars for at least 30 years [26] and they have recently partnered with Argo AI, who has funded their new autonomous vehicle research centre [27]. Bosch and University of Minho have launched several research and development projects concerning the future of transportation, with two of the latest ones (titled “Sensible Car” and “Easy Ride”) directly concerning autonomous driving [28].

1.3 Problem Statement

This work was developed within the “Sensible Car Automated Driving” project, and specifically the subproject concerning the development of a Road Condition Sensor (RCS). It improves upon an existing

microcontroller-based prototype that employs a highly sophisticated optical solution operating at near-infrared wavelengths, capable of detecting the presence (or absence) of water, ice and/or snow on the road. The basic theoretical principle that enables this type of sensor is that the surface of different materials have distinct light absorption characteristics. Thus, their presence can be identified by analysing the reflection spectrum and matching it with the signature of known substances. The prototype features 4 lasers of differing wavelengths (980 nm, 1330 nm, 1440 nm and 1550 nm) disposed at an equal distance from each other on the Printed Circuit Board (PCB), surrounding a photodetector that collects the diffuse reflection off the ground caused by the laser emissions. The analog signal coming from the photodetector goes through an analog front-end, before being sampled periodically using an Analog-to-digital Converter (ADC). Data values are collected from each laser, as they are fired successively one after the other (i.e. in a round-robin manner). These values are fed to a Support-Vector Machine (SVM) data classification algorithm which is run on the host Central Processing Unit (CPU), which then makes the classification result available to external systems via the Controller Area Network (CAN) bus. Additionally, a thermal management module was developed to perform laser frequency stabilization, ensuring the sensor's effectiveness in a wide range of ambient temperatures. Figure 1.1 presents a functional block diagram of the prototype described in this paragraph.

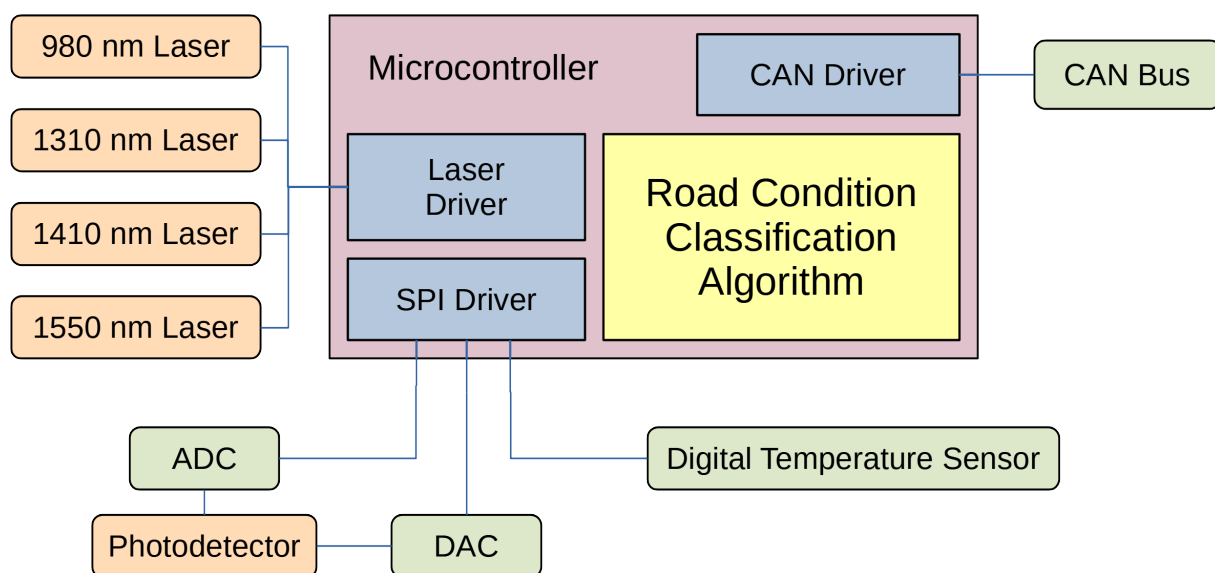


Figure 1.1: Block Diagram of the original prototype of the Road Condition Sensor.

However, the aforementioned prototype had limitations in terms of functionality and flexibility. The core issue lied in the inability to deploy the road condition classification algorithms to the selected host microcontroller. More specifically, the memory capacity of the system was not sufficient to store the data structures and intermediary values required to perform all of the computations on-board. By heavily

reducing the classification model's complexity and accuracy, it was possible to run data classification on the selected microcontroller but it was revealed that the real time metrics were not being met — that is, the ability to produce and communicate a new classification result via CAN every 10 ms.

1.4 Objectives

This thesis proposes a FPGA-centric implementation of the RCS prototype, which will be the target of a feasibility evaluation. The main goal is to gather empirical evidence to better determine the main strengths, weaknesses, opportunities and challenges in adopting an FPGA-based solution to perform the tasks that the microcontroller performs in the current prototype. A key realtime deadline must be met: new driving surface classification results must be delivered on the CAN bus once every 10 ms.

1.5 State-of-the-Art

The SVM data classification algorithm was identified as the critical feature of this thesis' work. It was imperative to demonstrate that an FPGA-based implementation with adequate accuracy was possible, while keeping resource usage to a minimum. Other functionalities of the system, while essential, did not pose as much of a technical challenge when migrating to the new platform — although analysing their integration in a high performance embedded system may prove more stimulating. Consequently, this Section focuses on defining the state-of-the-art regarding SVM implementations on FPGA devices. This research was largely supported by the work of Afifi et al. [29], who have produced an extensive survey on this topic citing numerous works. Here the focus is narrowed down to the implementation of the SVM classification phase, as the selected algorithm has been trained and evaluated prior to the realization of this work.

The implementation of SVMs on FPGA devices has been a fertile research area: various hardware architectures and designs have been proposed, and it is possible to identify some recurring trends in these research works. A subset of these trends is listed and analysed below, and evaluated in context with the requirements previously identified in this Chapter. The necessary background knowledge on SVM classifiers is provided ahead on Section 5.1.

Several of the designs proposed by researchers [30] [31] [32] made use of Systolic Arrays for their architectures. Systolic arrays are structures composed of identical computing nodes (commonly named Processing Elements (PEs)) that combine effective memory management with highly parallelized computations. These PEs can implement any type of operation, although the most common in the context of

SVM classifiers is the multiply-accumulate operation often used to compute the kernels. The systolic array relies on parallel data being fed to its inputs at regular intervals, an action that triggers the PEs to perform their computations in parallel — a computation scheme known as “transport-triggered architecture” [33]. Systolic arrays derive their name from the way data is pumped through several nodes in resemblance to the systolic action of the heart pumping blood through the cardiovascular system. Each PE computes a partial result, stores the result within itself and passes it downstream.

Dynamic Partial Reconfiguration (DPR) is the practice of reprogramming a portion of an FPGA at runtime without affecting the functionality of other parts of the device. This can be used to increase system flexibility and may also result in lower static power consumption, as a by-product of the reduced resource utilization. Some research works have employed this technique to also cover large-scale classification problems that cannot be implemented without hardware time-multiplexing [34] [35].

Some architectures offer a multiplier-less approach by implementing alternative kernels that provide iterative solutions based on shifts and adds to avoid using the scarce and “expensive” dedicated hardware multipliers, achieving great reductions in total design area [36] and sometimes minimal improvements in power efficiency [37].

Some researchers leveraged modern High-Level Synthesis (HLS) tools provided by FPGA device vendors that aid in reducing hardware development effort and time-to-market. Xilinx’s System Generator is a tool that allows for automatic code generation of systems modelled in Matlab/Simulink. Mahmoodi et al. [38] used this feature to produce SVM accelerators on hardware with near-identical classification accuracy when compared to their simulated counterparts. Xilinx’s HLS tools can enable even higher levels of abstraction by allowing embedded designers to model their system using C/C++ algorithms, massively shrinking development time for certain kinds of applications — Ning et al. [39] managed to implement a heterogeneous SVM system architecture in a single week, a mere fraction of the typical time when using traditional development flows which may take several months to conclude.

Another technique widely adopted is the use of custom fixed-point datapaths with different bit widths, as opposed to floating-point representations more commonly found in general purpose computers. Many designs achieved significantly lowered hardware usage with minimal reductions in classification accuracy [40] [41].

1.6 Thesis Structure

This thesis is structured as follows: Chapter 2 lists the research platform and tools used during the development of this thesis. It begins by explaining the platform selection requirements, with following Sections detailing the hardware and software used. Some key aspects of the selected FPGA architecture are highlighted, as their understanding proved critical for the design described in subsequent Chapters; then, Chapter 3 covers all the development stages of the principal data collection processes of the proposed design. A hierarchical FSM design is presented and discussed, and the main advantages and drawbacks of the design are discussed; an analysis of the auxiliary module for data pre-processing as well as clock domain crossing is provided in Chapter 4. The design and implementation of hardware division is exposed, as well as the handshaking mechanism used to load data to the new clock domain; in Chapter 5 a complete dissection of the SVM architecture that ultimately enables on-chip road condition classifications is provided. The entire process of conception is presented, from the reference software inference algorithms to the validation tests performed on the selected hardware. It also presents a possible complete RCS architecture, including all the modules necessary to achieve feature-parity with the previous prototype. The work required to produce and test this system is discussed, together with some considerations for future enhancement and validation of the design. Finally, Chapter 6 concludes this thesis and points out possible future research work to be conducted.

Chapter 2

Research Platform and Tools

This Chapter provides detailed documentation on the research platform and tools used during the development of this thesis. Section 2.1 presents the platform requirements which justify the selection of the ZYBO Z7-10 development board. Section 2.2 details the main hardware components that constitute the system proposed in this thesis. A particular emphasis is given on the Zynq-7000 System-on-Chip (SoC) architecture and some of its key features, as well as the set of sensors and actuators employed in the RCS. Section 2.3 presents the software tools used during the development of the system.

2.1 Platform Requirements

The ultimate goal of this thesis concerns an empirical study to evaluate the feasibility of an FPGA-based RCS prototype, by relying on the development of a subset of the prototypes' key features so as to collect the necessary qualitative and quantitative data. Taking this into consideration, the following requirements were identified and established:

- **Requirement 1:** The selected platform must feature an FPGA component.
- **Requirement 2:** The selected platform must enable scalable designs that minimize system re-factoring costs.
- **Requirement 3:** The selected platform must enable time-effective solutions.
- **Requirement 4:** The selected platform must be accompanied by robust development tools supporting simulation, validation, design synthesis, place-and-route and device programming.
- **Requirement 5:** The selected platform must be accompanied by ample documentation and technical support.

- **Requirement 6:** The selected platform must enable systems designs which rely only on programmable logic, software programmability or a combination of both.

These requirements define the use of a FPGA as the main data processing and control unit. The second requirement relates to the need of consolidating all of the features of the previous prototype into a single unit, while still providing a margin for future enhancements and feature-set expansion. Furthermore, the development time and funding of the project are both constrained, such that a suitable trade-off between development time and system cost had to be achieved. This limitation mandated the use of well-documented, widely adopted and mature FPGA platforms and development tools which could be obtained quickly and at a low cost. The final requirement concerns the fact that the discoveries produced on this thesis' work would be evaluated in comparison to other RCS prototype development projects that adopted different solutions and architectures. Thus, to ensure the different approaches could be evaluated as fairly as possible, a common platform capable of supporting all of them was needed.

2.2 Hardware Specification

This Section details the chosen hardware components which best suited the previously defined system objectives and requirements. An FPGA [42] is an integrated circuit that can be programmed on a hardware level, whereas standard CPUs derive their flexibility from a static hardware arrangement that can be programmed with software to perform various functions. FPGAs possess large amounts of configurable logic blocks and other miscellaneous components which can be routed together to implement complex circuits and computational architectures targetted for specific applications — without the need to develop and fabricate an ASIC, which is an extremely costly endeavour. These devices are unmatched for small-volume use cases requiring extreme flexibility and reconfigurability at a low-cost. Given an hypothetical FPGA containing an infinite number of programmable hardware resources, it is understood that it could solve any computable problem. As the amount of hardware resources necessary for this system could not be inferred, the selection of the development platform was guided mostly by monetary, flexibility and development time concerns.

2.2.1 Zynq-7000 SoC Family

The Zynq-7000 SoC [43] family of devices was selected to support this thesis' work. This line of products tightly integrates the software programmability of a dual-core ARM Cortex-A9 processor with the hardware programmability of a Xilinx 7-series Field Programmable Gate Array (FPGA) logic in a single

chip, delivering great design flexibility at a sensible cost. All Zynq-7000 devices share the same basic architecture, presented in Figure 2.1. The architecture is composed of two major sections: the dual ARM Cortex-A9 based Processing System (PS) and the 7-series equivalent Programmable Logic (PL).

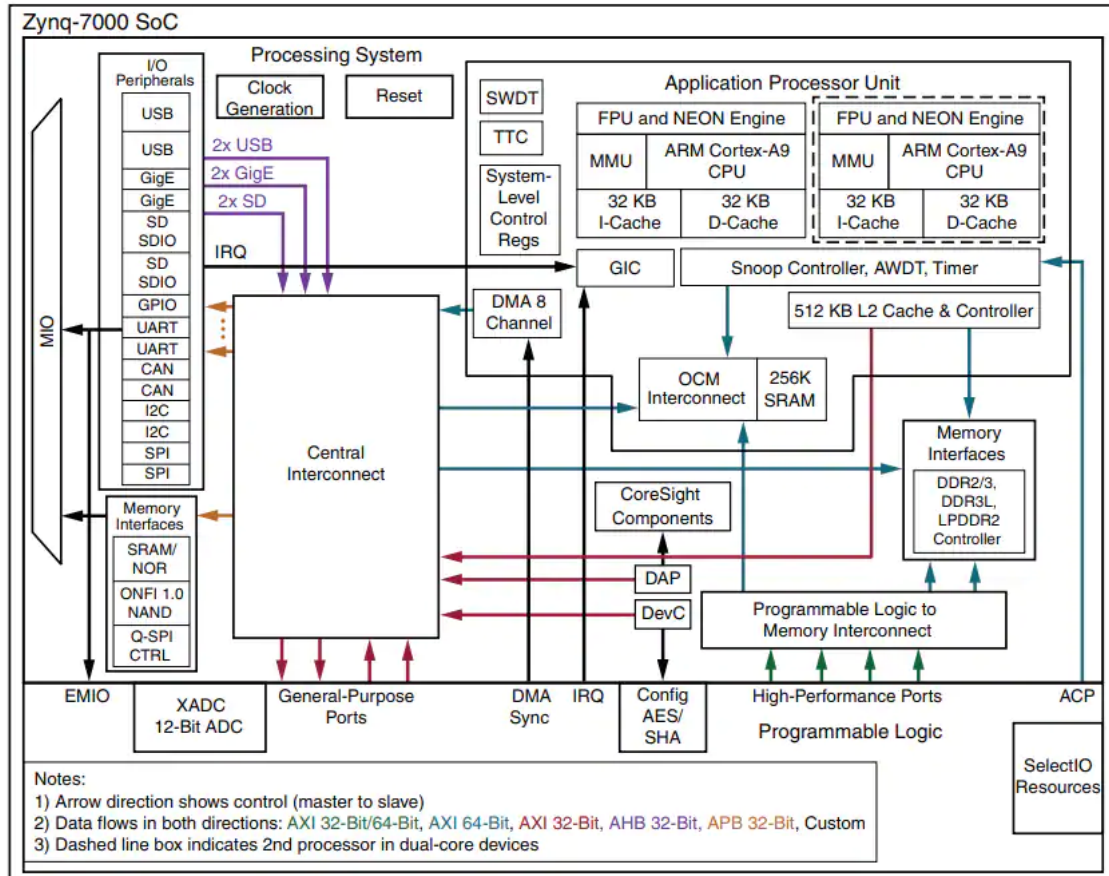


Figure 2.1: Zynq-7000 SoC Architectural Overview

The nucleus of the PS is the Application Processing Unit (APU), which combines several computation units as well as peripheral interfaces, cache memory, and memory interfaces. The main computational units are the aforementioned two 32-bit ARM Cortex-A9 cores, capable of operating at frequencies up to 1 GHz on the higher speed grade devices. Each core's functionality has been extended with the inclusion of a separate hardware unit: an advanced Single Instruction, Multiple Data (SIMD) coprocessor named ARM NEON™, accompanied by a Vector Floating Point (VFP) unit which together provide major acceleration of Digital Signal Processing (DSP) and media-processing algorithms. Each processor has its own set of self-contained 32 KB L1 caches for instructions and data, which are complemented by a shared 512 KB of external L2 cache - cache coherency is assured by the Snoop Control Unit (SCU). An additional 256 KB of on-chip Static Random-Access Memory (SRAM) and a Direct Memory Access (DMA) unit round out this summary of important APU features. The PS supports multiple memory technologies, by virtue of

its combination of Quad-SPI, Double Data Rate (DDR) Synchronous dDynamic Random-Access Memory (SDRAM) and static memory controllers. The input/output peripherals are a collection of industry-standard interfaces for external data communication, containing a rich set of both high-bandwidth ($2 \times$ Gigabit Ethernet, $2 \times$ USB 2.0, $2 \times$ SDIO) and low-bandwidth modules (General-Purpose Input/Output (GPIO), $2 \times$ SPI, $2 \times$ UART, $2 \times$ CAN, $2 \times$ I²C).

The PL provides a rich set of user-configurable hardware blocks based on 7-series FPGA logic. These include Configurable Logic Blocks (CLB), 36 Kb RAM blocks, DSP48E1 slices, clock management units and configurable I/O. All Zynq-7000 devices include the Xilinx Analog-to-Digital Converter (XADC), a basic building block that enables Analog Mixed Signal (AMS) functionality, facilitated by the inclusion of on-chip sensors that support measurement of the on-chip power supply voltages and die temperature. Some features, such as low-power Gigabit transceivers and integrated interface blocks for Peripheral Component Interconnect (PCI) Express designs are reserved for the more expensive Zynq-7000 parts.

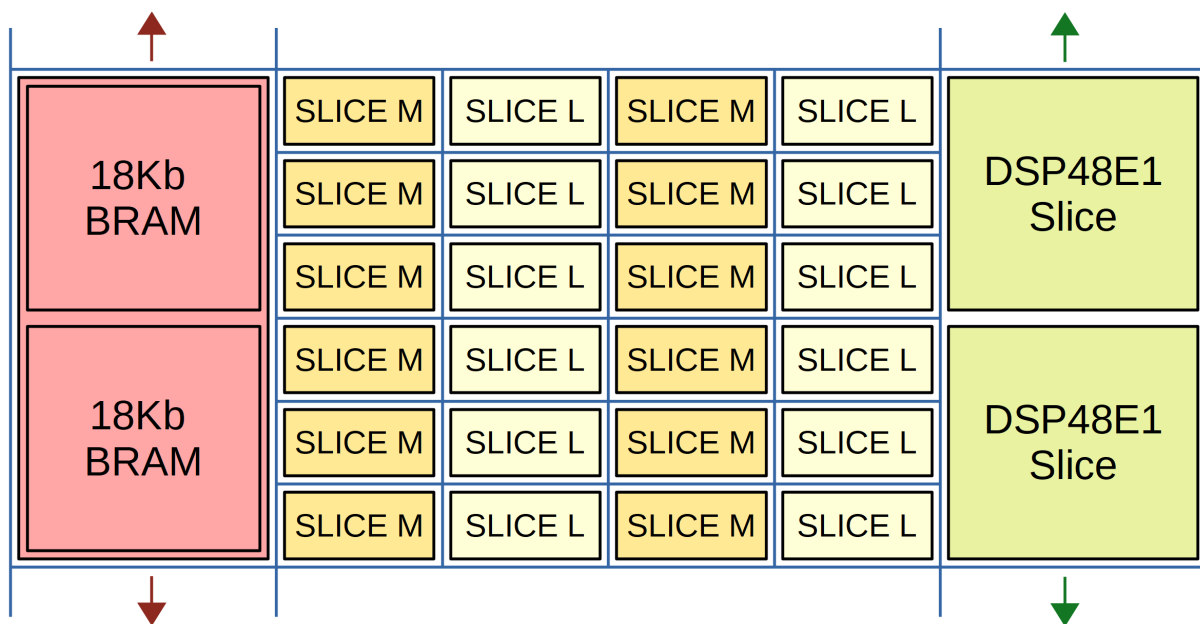


Figure 2.2: Detail of the Zynq-7000's column-based PL layout.

The Zynq-7000's PL is an implementation of the Advanced Silicon Modular Block (ASMBL) architecture developed by Xilinx (represented in Figure 2.2). This architecture has enabled them to create varied PL parts with distinct resources in different quantities to better suit certain applications, while minimizing engineering efforts. The configurable logic blocks are distributed along columns, which has some implications for optimizing designs for deployment on FPGAs. For instance, the memory blocks (described in Section 2.2.2) are stacked in a column that is in close proximity to the DSP slices (described in Section 2.2.3), which ensures that algorithm coefficients can be stored close to where they will be needed.

The Zynq-7000 devices make use of the ARM Advanced Microcontroller Bus Architecture (AMBA) as the on-chip interconnection standard to connect and manage the functional blocks within the SoC design. More specifically, the interconnect in these devices use the AMBA Advanced eXtensible Interface (AXI) version 4 as the main protocol for data communications [44]. AXI is a point-to-point interconnect that employs multiple dedicated channels for reading and writing bursts of data, between a single Master and a single Slave device. The AXI interconnect enables multi-master, multi-slave AXI systems and is therefore a vital component of the Zynq-7000 architecture.

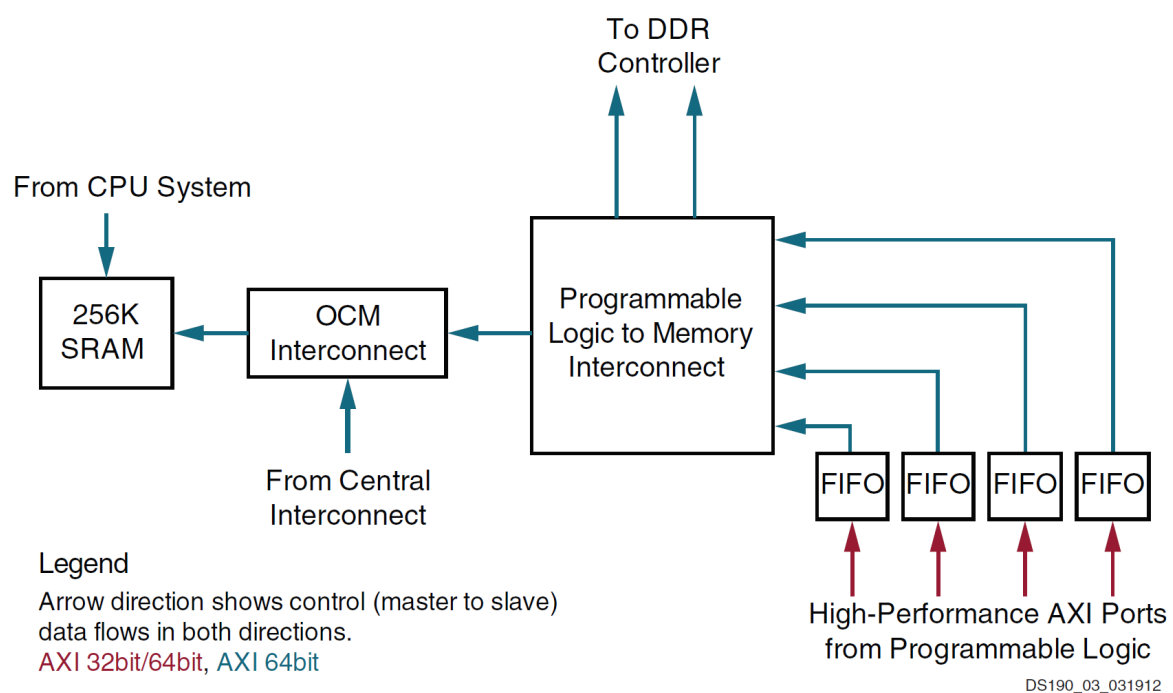


Figure 2.3: PL Interface to PS Memory Subsystem.

Besides bridging several PS components, the AXI interconnect enables interaction with the PL. The main component of the PS-PL interface is a set of nine AXI interfaces, which are configured to serve different purposes: two general purpose 32-bit AXI Master interfaces, two general purpose 32-bit AXI Slave interfaces, four high-performance 32-bit AXI Slave ports with direct access to DDR memory and on-chip memory (as shown in Figure 2.3, adapted from DS190 [43]) and a 64-bit AXI slave Accelerator Coherency Port (ACP) for coherent access to CPU memory from the PL. The primary PS-PL interface is complemented by some additional miscellaneous signals that comprise a total of 16 interrupts, DMA, debug interfaces as well as clock as reset signals.

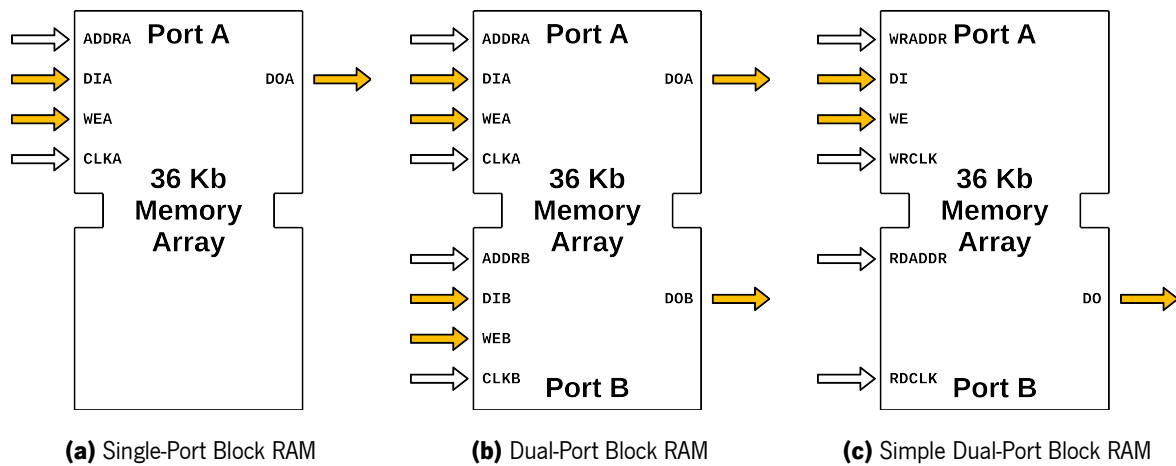


Figure 2.4: Multiple supported Zynq-7000 BRAM configurations.

2.2.2 Block RAM

Every Zynq-7000 SoC device has between 60 and 755 dual-port block RAMs, where each can be configured as either two independent 18 Kb RAMs or one 36 Kb RAM. Each of these memory blocks (commonly referred to as Block RAM (BRAM) in Xilinx nomenclature) comes with two completely independent ports with individual address, clock, write enable and clock enable signals, making them a flexible data storage solution. These memory units support multiple access configurations, including: Single-Port Block RAM (Figure 2.4a), which allows for sequential read and write operations through a single read/write port; Dual-Port Block RAM (Figure 2.4b), which provides two independent ports each capable of reading and writing to memory, and capable of operating simultaneously; and Simple Dual-Port Block RAM (Figure 2.4c), defined as having one read-only port and one write-only port with independent clocks.

The memory content can be initialized or cleared by the configuration bitstream, facilitating the use of BRAMs as static Read-Only Memory (ROM) if necessary. Dedicated logic in the BRAM allows for the implementation of fully-featured synchronous or dual-clock (asynchronous) First In, First Out (FIFO) data structures, eliminating the need for additional general fabric resources to implement the same functionality. This built-in support comes in the form of embedded read and write pointers, common FIFO flags like “Full” and “Empty” as well as programmable “Almost Full”, “Almost Empty” and read/write error flags.

The 36Kb BRAMs are endowed with built-in Hamming Error Correction Code (ECC) functionalities. These are available only in the simple dual-port or FIFO mode, when configured for 512 data entries of 64-bits each. Eight protection bits are generated during each write operation, and used during each read operation to correct any single-bit error, or to detect (but not correct) any double-bit error.

It is possible to create larger RAM structures thanks to dedicated resources built for minimal routing

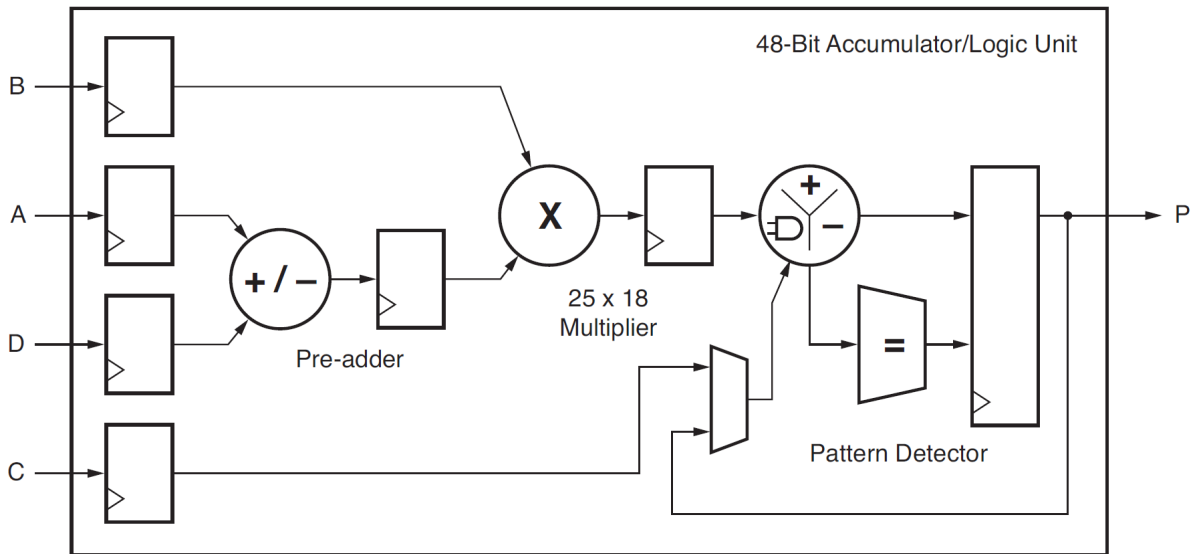


Figure 2.5: Basic DSP48E1 Slice Functionality.

delays. Two vertically-adjacent 36 Kb BRAMs can be cascaded to form a single 64 Kb storage unit without using any external resources. With the aid of external configurable logic, both wider as well as deeper memories (composed of multiple BRAMs) can be instantiated in the design, with minimized performance penalties.

Flexibility is further enhanced by enabling independent read and write port width settings. For example, each 36 Kb BRAM can be configured for 1-bit, 2-bit, 4-bit, 9-bit, 18-bit, 36-bit or 72-bit width data entries in simple dual-port mode. In addition, the read port width can differ from the write port width. Certain features, such as the built-in ECC and dedicated FIFO logic, constrain the number of valid port width configurations. Additional documentation on the Zynq-7000 BRAM memory unit can be consulted on UG473 [45].

2.2.3 DSP48E1 Slice

All Zynq-7000 devices come equipped with the same DSP48E1 cell, commonly referred to as DSP slice, with numbers ranging from 80 up to 2020 slices on top-shelf components. These DSP slices support many common functions, which include multiply, add-multiply, multiply-accumulate and tree-input add operations, as well as bitwise logic functions on two 48-bit binary numbers. These operations are selected through control signals, some of which can be set on the fly which provides added design flexibility by dynamically changing DSP48E1 functionality from clock cycle to clock cycle. A simple representation of this computational unit is shown in Figure 2.5, adapted from UG479 [46].

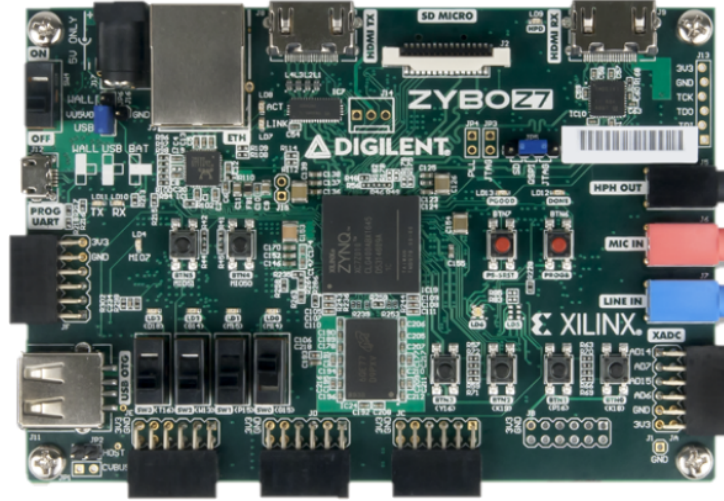


Figure 2.6: ZYBO Z7-10 Board.

On the left side of the image, one can see the elements which handle input conditioning, in the form of a pre-adder and input pipeline registers. Numerous programmable pipeline registers (for input operands, intermediate products, and accumulator outputs) exist in several places within the DSP48E1 slice, to enhance throughput in pipelined applications. These structures are followed by a 25×18 multiplier (which accepts signed two's complement operands) followed by multiplexers and a three-input adder/subtractor/accumulator. The pattern detector at the output of the DSP slice can detect if the output of the DSP48E1 slice matches a pattern, providing support for convergent rounding and overflow/underflow/saturation detection of the accumulators. If wider functionality is required, numerous cascade paths are available for both inputs and outputs, allowing for multiple DSP slices to be chained vertically to implement more complex functionality. This method makes use of dedicated resources rather than general fabric routing resources, increasing performance while also reducing power consumption. This technique is limited by the number of DSP slices available per column as well as the height of the DSP columns, all of which vary from device to device. Additional documentation on the Zynq-7000 DSP48E1 slice can be retrieved from UG479 [46].

2.2.4 ZYBO Z7-10

The development board selected for the realization of this thesis is the ZYBO Z7-10 (Figure 2.6), as it fulfils all the previously mentioned platform requirements. The ZYBO Z7-10 [47] is a low-cost development board based on the previously described Xilinx Zynq-7000 All Programmable SoC, manufactured by Digilent.

This development board contains the smallest Zynq device of the Zynq-7000 SoC family, the XC7Z010-

1CLG400C. The PS is identical to those found in other Zynq-7000 family devices, with the sole limitation being the maximum frequency, which is limited to 667 MHz in this variant. The biggest trade-off when selecting a lower-end part comes in the vastly reduced programmable logic resources — which directly influences other system metrics such as total memory available and maximum parallel processing performance. It features a Artix-7 equivalent programmable logic section comprised of 4,400 configurable logic slices, 17,600 LUTs, 35200 FFs, 60 blocks of 36 Kb BRAMs (adding up to a total of 2.1 Mb in memory capacity) and 80 DSP slices.

The development board incorporates a varied array of peripherals, memory modules and other utilities which make it an excellent prototyping platform. It is particularly well suited for embedded computer vision applications, as evidenced by the MIPI CSI-2 compatible Pcam connector and pair of HDMI sink and source ports. Audio capabilities are also included, in the form of a dedicated low power audio codec chip that is connected to stereo headphone, stereo line-in, and microphone jacks. The Zybo Z7-10 includes a 1 GiB DDR3L module capable of memory interface speeds as high as 533 MHz, connected to the PS DDR memory controller. It possesses an additional 16MB Quad-SPI Flash as well as a microSD slot, both of which can be used to boot the Zynq if the corresponding boot mode is selected on power-on. Dedicated transceiver chips and PHY components for both Gigabit Ethernet and USB 2.0 (which can act as an embedded host or a peripheral device) are included. Six push-buttons, four switches, five LEDs and a single RGB Light-Emitting Diode (LED) complete the list of components that come equipped in this development board. Further expansion can be achieved by populating the six 2×6 pin Pmod ports, which fall into one of four categories: MIO connected (acronym for Multiplexed Input/Output, accessible only from the PS peripheral controller cores), XADC (wired to the auxiliary analog input pins of the PL), high-speed (ideal for high speed differential signals) or standard (equipped with series resistors for short circuit protection, which in turn limit switching speeds).

2.2.5 Sensors

The hardware that comprises the data acquisition solution was not developed within the scope of this thesis. However, it was necessary to understand this system in order to produce configurable logic modules capable of interfacing with it. The data acquisition system employed by the previous RCS prototype includes the ADS8332 analog-to-digital converter manufactured by Texas Instruments, Analog Devices' ADT7301 digital temperature sensor as well as an NTC Thermistor produced by Murata Manufacturing. A DAC8162 Digital-to-Analog converter by Texas Instruments [48] was also built into the previous prototype's PCB, and more specifically in the amplification stages of the photodetector circuit. Its role in the system is not

explored for this work, but was still selected as a potential target for interfacing with a generic SPI Module. This document does not include a comprehensive examination of the amplification, filtering and signal conditioning circuits which ensure the proper functioning of each sensor, as these do not fit this thesis' scope.

2.2.5.1 Analog-to-Digital Converter

A simple functional block diagram of the selected analog-to-digital converter, the ADS8332 [49], is shown in Figure 2.7. The sensor is coupled to the analog front-end of an external photodetector which detects light in the near infrared range (from 950 to 1600 nm).

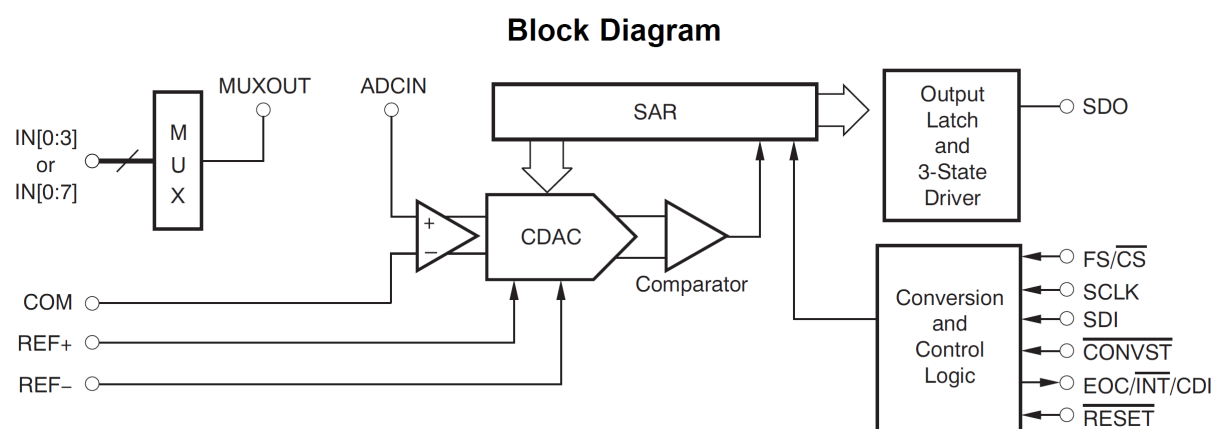


Figure 2.7: ADS8332 Functional Block Diagram

The ADS8332 is a SAR ADC, often used in embedded applications due to their low power consumption, high resolution and accuracy, and a small form factor when compared to other ADC architectures. The typical architecture of these devices include a sample-and-hold structure (not explicitly represented in Figure 2.7, as the capacitive DAC employed by the ADS8332 inherently implements this functionality), an N-bit search DAC, an analog comparator and the Successive-Approximation-Register register. This kind of ADC implements a binary search algorithm composed of two phases: the acquisition phase and the conversion phase. During the acquisition phase, the analog input signal is connected to the capacitive DAC, during which the sampling capacitors will charge from an initial voltage to the same voltage of the input signal. This implies a minimum setting time for the stored voltage to stabilize, which must be respected to ensure accurate conversion results. When the acquisition phase ends (in the case of the ADS8332, this is user-triggered by an input "start conversion" signal), the sample-and-hold mechanism is disconnected from the external circuit. This coincides with the start of the conversion phase, where the sensor's control logic combines the capacitive DAC, the analog comparator and the SAR to implement

a binary search algorithm. Starting with the MSB, each sampled bit value is determined based on the comparison between the input voltage and a DAC reference voltage. This mechanism implies again a minimal conversion time, that is a function of the conversion clock frequency and the number of bits to be determined. These timing requirements limit the maximum sampling rate, which is one of the main drawbacks of SAR ADCs when compared to other types of converters.

This sensor in particular offers a total of 8 unipolar input channels which are fed to a multiplexer. The multiplexer's output is one of two inputs for the internal conversion circuitry, the other being a common pin usually connected to ground. The device features two channel selection modes: Manual mode, where data commands are used to access the desired channel; and Auto mode, in which a selection of input channels is sampled and converted continuously in a fixed order. The “heart” of the device is the 16-bit SAR, which uses an external voltage reference ranging from 1.2 V to 4.2 V and is capable of performing up to 500 kSPS. The converter can be clocked by either an internal oscillator or an external clock signal based on the serial interface clock. Internal registers, accessible by the serial interface, facilitate device configuration and data readouts. Other digital interface signals include an asynchronous reset input signal, a input signal to trigger new data conversions and a programmable status output signal useful for detecting when new data is available for retrieval.

2.2.5.2 Temperature Sensors

The ADT7301 digital temperature sensor [50] produced by Analog Devices (see Figure 2.8) was used in the RCS prototype for board temperature monitoring.

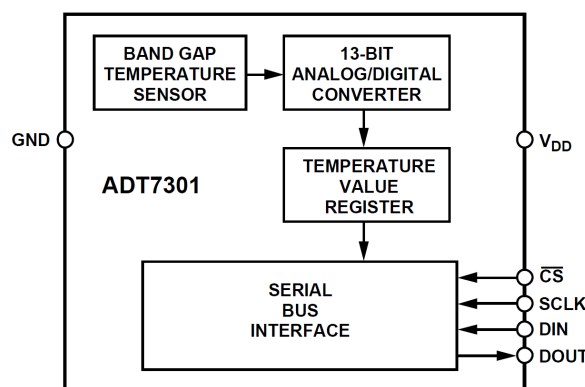


Figure 2.8: ADT7301 Functional Block Diagram

Its main sensing element is the internal band-gap temperature sensor (rated for operation at temperatures as low as -40 °C and as high as +150 °C) which is coupled to a 13-bit ADC, producing data readouts at a resolution of 0.03125 °C. It possess a flexible serial interface that supports the SPI, Quad-SPI and

MICROWIRE communication protocols and can also interface directly with DSPs.

The other sensor is an NTC 10 k Ω (at 25 °C) thermistor, rated for operating temperatures as low as -40 °C and up to 150 °C. A thermistor is a type of resistor whose resistance is heavily correlated with its temperature, assuming an approximately linear relationship between the two — for Negative Temperature Coefficient (NTC) thermistors, resistance decreases as temperature rises.

2.2.6 Actuators

The design and implementation of the actuation system, in similarity with the previously described sensors, was already undertaken and finalized before this thesis' work had commenced. It remains necessary to study these components to gather the exact functional requirements of the interfacing systems to be developed within the scope of this thesis.

The prototype features a sophisticated optical system, in the form of a laser diode array placed on the PCB side by side in a circular arrangement. Each laser emits a different nominal wavelength in the near-infrared region. The PCB features dedicated amplification circuits for each laser, each accepting a single digital input to toggle the respective diode on or off.

The thermal control module features active cooling in the form of the CP30238H Peltier module [51] from CUI Devices. A Peltier module is a thermoelectric cooler (therefore requiring no circulating cooling fluid or moving parts) which produces temperature differential on each side (while one side gets warmer, the other gets cooler) as a function of the voltage applied on its terminals. The removal of heat from the Peltier's warmer side was initially handled by an external assembly combining a heat sink and fan in earlier RCS prototypes, which was later deprecated as the convective heat transfer (facilitated by the vehicle's motion) was deemed sufficient.

2.3 Software Specification

When developing embedded systems, a robust set of development tools with ample documentation and technical support can be just as crucial for the project's success as the specific hardware specifications and features. This is particularly relevant when developing for FPGA platforms, which are decidedly a niche product when compared to the wide adoption of microcontrollers and microprocessors in many scientific endeavours and industry applications. This widespread use has resulted in the proliferation of readily-available, well documented and supported development tools for those devices. In contrast, the development platforms and specific toolchains available to program FPGAs are often vendor-locked,

a problem that is compounded by the relative scarcity of free and/or open-source tools. Besides that, many other essential development tools like hardware simulators and numeric analysis tools are often expensive to acquire. The thesis' work was developed mostly using the design tools made available by Xilinx, the manufacturers of the Zynq-7000 SoC. These are the Vivado Design Suite [52] and the Vitis software development platform [53].

Vivado Design Suite is a set of tools developed by Xilinx for synthesis and analysis of Hardware Description Language (HDL) designs, which primarily targets development of embedded firmware for Xilinx FPGA products. It was developed as a replacement to their now discontinued Xilinx ISE (acronym for Integrated Synthesis Environment) set of development tools with additional features for SoC development. It comes with a built-in logic simulator, allowing for the creation of test bench programs in HDL. A typical test bench produces adequate input stimuli for the hardware component being tested (known as the Device Under Testing (DUT)), which can later be viewed together with other signals for manual validation. Xilinx' proprietary HDL synthesizer (support both Verilog and VHDL languages) is included, as well as automated place-and-route procedures and device programming utilities. Vivado High-Level Synthesis (HLS) tools enable C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to write code with Register-Transfer Level (RTL)-abstraction level languages like Verilog and VHDL. The Vivado IP Integrator can be used to accelerate development of complex systems by providing a simple graphical interface where blocks of logic or data (named Intellectual Property (IP) cores in Xilinx nomenclature) can be quickly integrated and configured. Xilinx provides a big library of IP cores ranging in sophistication from simple logic and arithmetic operators to complex processors and co-processors, filters and memory systems. Vivado supports Xilinx's 7-series and all the newer Xilinx devices (UltraScale and UltraScale+ series).

Xilinx also provides clients with Vitis, a set of tools built for application acceleration and embedded software development. Vitis is a set of older separate tools which are now available and distributed in a single package: the Xilinx Software Development Kit (XSDK); the Software-Defined System On Chip (SDSoC) environment and the Software Defined Acceleration (SDAccel) development environment. Vitis can be used to program the hard ARM Cortex-A9 cores present in Zynq-7000 series devices, as well as soft cores that may be instantiated in the PL.

Chapter 3

Lasers and Photodetector Controller

The Road Condition Sensor's light sensor and its set of actuators provide the foundation for sophisticated data classification algorithms to be implemented successfully. The system's data collection process must be carefully designed and implemented, as no relevant insights can be derived from inaccurate or insufficient data.

This Chapter details the system's data collection units and processes. Section 3.1 details the desired capabilities and characteristics of the system, which guided the subsequent development stages. Section 3.2 and Section 3.3 examine the design and implementation phases of development, respectively. Each of these Sections is further divided into subsections devoted to a single component, thus describing the entire sensory architecture from bottom to top. The latter Section exposes and discusses objective metrics of the final system, such as FPGA resource utilization, latency and throughput, among others. Section 3.4 reflects upon the system's initial objectives and its ultimate achievements, identifying key issues and opportunities for improvement.

3.1 Analysis

The μ C-based RCS prototype that predates this thesis' work features a sophisticated optical solution which is at the core of the system's road condition evaluation capabilities. The main components composing that system have been succinctly described in the previous Chapter, more specifically in Sections 2.2.5 and 2.2.6. From those Sections a set of functional requirements can be extrapolated, which dictate the minimum feature set of the data collection mechanisms that must be designed and implemented for the RCS:

1. Perform communications with the converter via SPI, with the ability to read from and write to the device. Must abide to all timing, electrical and functional requirements of the device to ensure correct behaviour;

2. Reset the converter to a known default state. This should precede any configurations or data readout activities to ensure the processes are reliable and produce repeatable results;
3. Perform data readouts at a constant time period, ensuring a steady supply of data for the ensuing data processing circuits, which must be notified when a new data samples become available;
4. Update the lasers' state in synchrony with the data readout operations, ensuring an ADC sample is produced and collected for each of the four wavelengths of diodes.

Additionally, it is possible to highlight some desirable properties which, if integrated, would enhance the data retrieval unit and the system as a whole. The author identifies the following non-functional requirements:

1. Fully synchronized system implementation, by ensuring the whole design exists within a single clock domain. This would remove the need to develop additional circuitry to account for asynchronous events which could compromise the necessary real time capabilities of the RCS;
2. A design favouring a high degree of configurability and adaptability, to facilitate further evolutions of the system. This quality is of extreme value for an multi-year academic research project in a continuous process of revision, which requires rapid iteration and the ability to quickly introduce new features in a design.

3.2 Design

This Section details the specific configuration and interactions implemented on the new system, citing the relevant technical information and considerations which guided the design process. The proposed solution is built upon an hierarchical FSM structure with a total of three nested modules. The design of each layer is detailed below in independent Sections, in a bottom-up order: Section 3.2.1 details the SPI Module which implements a flexible SPI-compatible interface capable of reading and writing to external components; Section 3.2.2 that details the module which handles ADC initialization and performs data readouts upon request; and finally Section 3.2.3, which triggers data readouts in synchrony with laser commutations, and makes the data samples available to the data pre-processing algorithms described in Chapter 4. The development of the previous RCS prototype, as well was the work shown in this thesis, is part of a research and development initiative from which closed patents will be derived. As such, some particular data collection system's timing requirements and configurations can not be disclosed in this

document, and were not made available to the author. This constraint was dealt with by making designs more flexible, so as to ensure the proposed system can replicate many possible configurations.

3.2.1 SPI Module

The selected ADC module includes a Serial Peripheral Interface (SPI) for programming, and a compatible interface for the FPGA was needed to interact with the device. The SPI protocol is a synchronous, full-duplex serial communication bus interface specification, which supports a single-master, multiple-slave architecture. This means that synchrony is assured by a clock signal generated by the Master device, while the “full-duplex serial” categorization implies that data is serialized bit-by-bit through two data wires. The interface comprises a total of four signals:

1. Serial clock (SCLK) is a master output clock signal with respect to which the SPI devices transfer data. The original Motorola standard specifies configurable Clock polarity (CPOL) and Clock phase (CPHA) for a total of four possible configurations, which must match between master and slaves devices;
2. Chip Select (CS), or SS (Slave Select), is a master output signal which allows for the selection of an individual slave SPI device, while slave devices that are not selected do not interfere with bus activities. A master device may include several of these signals (as seen in Figure 3.1), and it is commonly a active-low signal;
3. Master Output, Slave Input (MOSI) is the signal used to serialize data from the master to the slave device(s);
4. Master Input, Slave Output (MISO) is the signal used to serialize data from the slave to the master device.

This communication protocol is what is known as a “*de facto*” industry standard, which means it has received wide adoption in many applications but there is a lack of formal standardization. While a standard was initially developed and published by Motorola in 1979 [54], the reality is that each device has its own particular adaptation of SPI with slight but important differences. For this reason, it is necessary to carefully analyse the interface requirements of each targetted device — for instance, the ADS8332 supports both 4- and 16-bit long SPI transmissions, while the ADS7301 works with 14-bit transfer lengths.

The selected ADS8332 converter’s interface specifications are summarized below in Table 3.1, adapted from [49]. These are valid for a digital reference voltage of 2.7 V, and only timing requirements

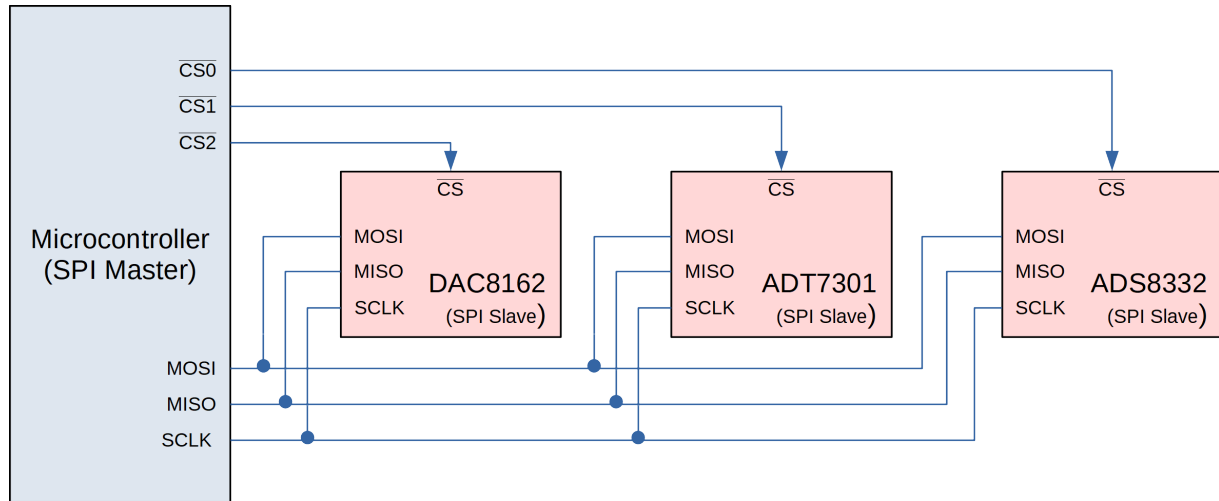


Figure 3.1: SPI bus architecture of the previous RCS prototype.

relevant to the adopted solution are portrayed in the Table. An accompanying timing diagram is shown in Figure 3.2, adapted from [49].

The ADS8332 be programmed to run conversions using the external SCLK instead of the internal oscillator, allowing system designs with total system synchronization. This design choice satisfies one of the non-functional requirements, which streamlines the design efforts of other system modules. The time interval between each new data acquisition can be known *“a priori”*, allowing for the implementation of a control system that abides to hard realtime metrics, as it is not dependant on asynchronous signals coming from the ADC. The resulting Conversion clock (CCLK) frequency is given by the formula:

$$f_{\text{CCLK}} = \frac{f_{\text{SCLK}}}{2}$$

In this configuration, the minimum SCLK period is 47.6ns which translates to a maximum SCLK frequency of roughly 21 MHz — however, the range of frequencies supported by the proposed design goes from 1 MHz up to 20 MHz. It was determined that this lower maximum frequency facilitated implementation efforts and did not significantly impact overall system performance or compromise any target realtime deadlines. With the goal of reducing the overall system’s development time and effort, it was intended that this single generic SPI module could be used as a foundation to interact with all SPI-compatible devices present in the system — specifically, the ADS8332, the ADT7301 and the DAC8162. As the last two are not performance-critical, enabling the highest possible serial communication with those devices was not a requirement. However, slight modifications must be made as each module’s clock signal polarity and phase configurations differ. A functional block diagram of the designed SPI Module is presented in Figure 3.3.

Name	Description	MIN	NOM	MAX	UNIT
t_{SCLK}	Serial clock (SCLK) period	47.6	-	1000	ns
f_{CCLK}	Conversion clock (CCLK) frequency	0.5	-	10.5	MHz
t_{WH1}	Pulse duration, CS high	40	-	-	ns
t_{SU3}	Setup time, CS falling edge to first SCLK falling edge	8	-	-	ns
t_{D1}	Delay time, SCLK falling edge to SDO invalid	5	-	-	ns
t_{D2}	Delay time, SCLK falling edge to SDO valid	-	-	20	ns
t_{D3}	Delay time, CS falling edge to SDO valid	-	-	20	ns
t_{D4}	Delay time, CS rising edge to SDO 3-state	-	-	10	ns
t_{SU4}	Setup time, SDI to SCLK falling edge	8	-	-	ns
t_{SU5}	Setup time, last SCLK falling edge before CS rising edge	10	-	-	ns
t_{SU6}	Setup time, SCLK rising edge to CS rising edge	10	-	-	ns
t_{H3}	Hold time, SDI to SCLK falling edge	8	-	-	ns
t_{H4}	Hold time, last SCLK falling edge before CS rising edge	2	-	-	ns
t_{H5}	Hold time, SCLK rising edge to CS rising edge	2	-	-	ns

Table 3.1: Summary of ADS8332 serial interface timing requirements.

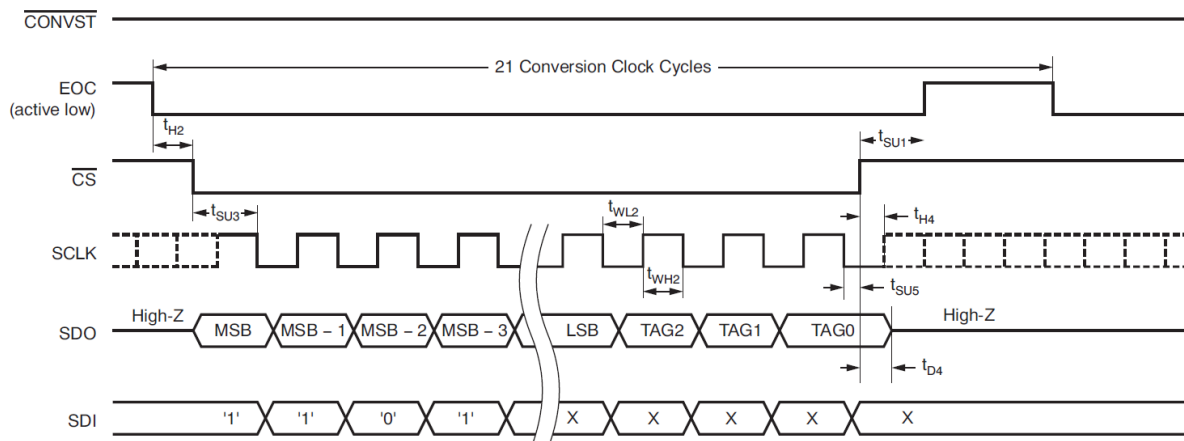


Figure 3.2: ADS8332 serial interface timing diagram.

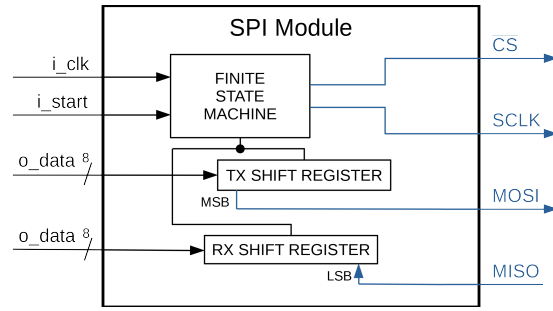


Figure 3.3: SPI Module Functional Block Diagram.

A Moore Finite-State machine (FSM) was design to implement the desired SPI module functionality, shown below in Figure 3.4. Each state has a set of associated output signals:

1. The negative logic chip select output signal, “o_cs_n”;
2. The serial clock output signal, “o_sclk”;
3. The active-high auxiliary signal “o_txbuf_write”. When this is signal is active, the value of an internal transmission buffer is updated;
4. The active-high auxiliary signal “o_txbuf_shift”. When this is signal is active, the value of an internal transmission buffer is left-shifted;
5. The active-high auxiliary signal “o_rxbuf_shift”. When this is signal is active, the value of an internal reception buffer is left-shifted;
6. The active-high auxiliary signal “o_txbuf_write”. When this is signal is active, the value of an internal reception buffer is loaded into an output buffer containing the latest data read from the serial interface.

The signals “o_txbuf_write” and “o_txbuf_shift” implement a Parallel In, Serial Out (PISO) block that receives the parallel data from the ADS8332 Controller Module and then shifts it out at a given serial clock rate, which is derived from the division of the input clock by four (see Equation 3.1).

$$f_{\text{SCLK}} = \frac{f_{\text{i_clk}}}{4} \quad (3.1)$$

Similarly, the signals “o_rxbuf_write” and “o_rxbuf_shift” implement a Serial In, Parallel Out (SIPO) block that handles SPI message reception by serializing the MISO input. The received data is exposed in an output port connected to the ADS8332 Controller Module.

A state transition diagram for the FSM design is presented in Figure 3.4. Each state includes additional information on its Moore outputs, i.e. the output values which depend only on the current state of the automaton. The purpose of each output and auxiliary variable represented is further explained in accompanying paragraphs. This scheme (inspired by the UML statechart notation [55]) helps in describing the intended functionality with greater granularity, and is reused later in Figures 3.5 and 3.6 to represent other FSMs used in the system.

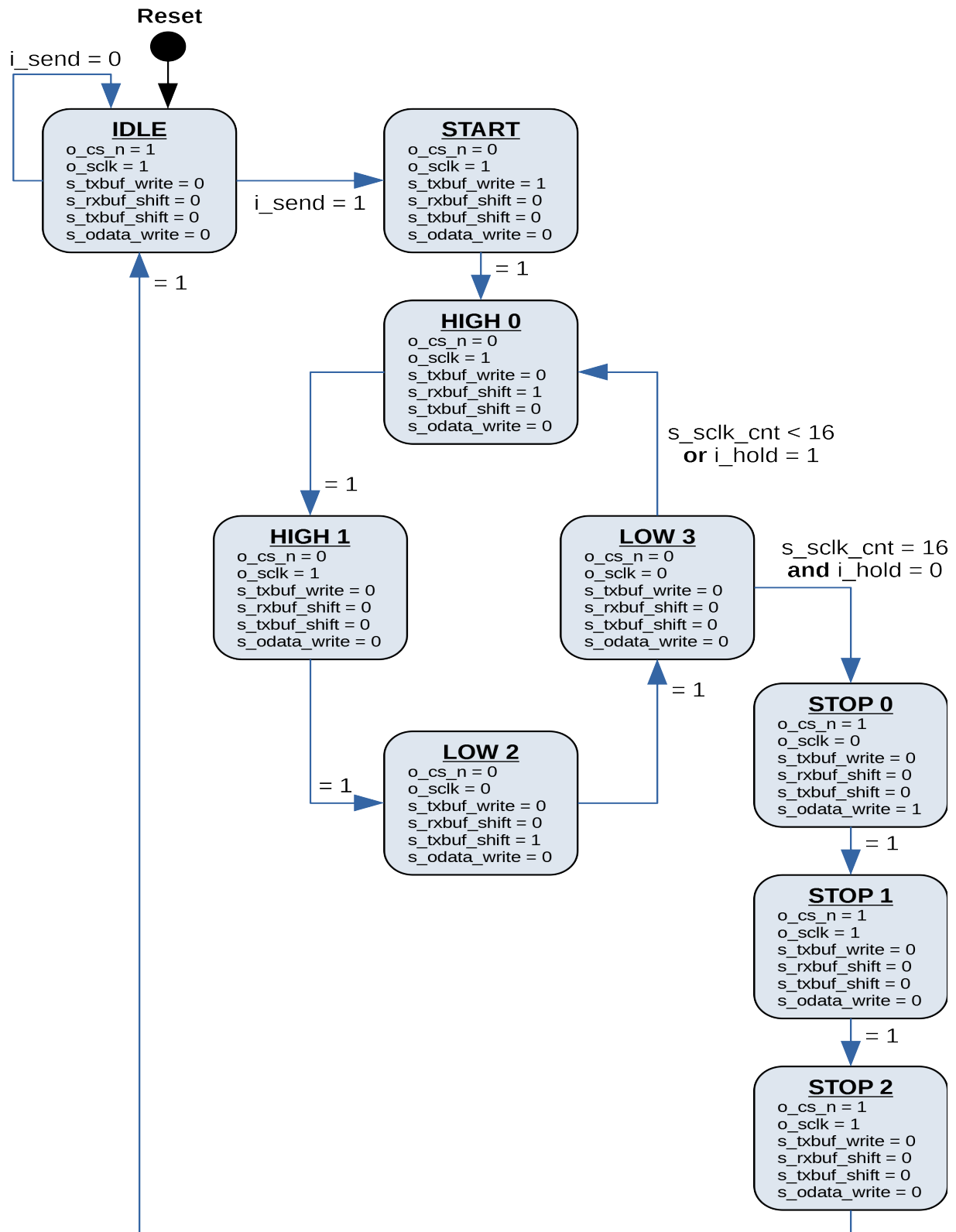
The entry point of the state machine (denoted by a filled black circle) is the “IDLE” state, where both the SCLK and the CS interface lines are held at a default “high” logic value. This corresponds to a CPOL=1, CPHA=1 configuration where the output signals change on the rising edge of SCLK, and the input signal is sampled on the falling edge for both slave and master devices. When the transmission/reception request signal “o_rxbuf_write” is active, the automaton transitions to the “START” state where CS is toggled low to mark the beginning of the communication window. From there, a loop of four states is entered which implement the serial clock cycles, and where data transmission and reception is performed. This loop is exited on the “LOW 3” state after 16 clock cycles have elapsed, after which a sequence of “STOP” states is entered where the communication window is closed. Alternatively, the SCLK cycle loop can be held indefinitely by an upper module if the “i_hold” signal is held high. The final two “STOP” states are identical, and implement the necessary time delay to abide by the ADS8332 timing requirements previously identified in Table 3.1. The final implementation details of this module are exposed in Section 3.3.1.

3.2.2 ADS8332 Controller Module

The next module in the hierarchy is the ADS8332 Controller Module. As its name would suggest, it performs ADC initialization as well as data readouts by relying on the previously described SPI Module for serial communications.

The converter may be programmed via its digital interface, which adheres to the SPI protocol and grants access to a single internal 16-bit register. This internal register is subdivided in two parts: the first four bits comprise the Command register (CMD), while the latter twelve bits contain the Configuration register (CFR). The digital interface is complemented with signals for resetting the converter, starting the internal conversion process as well as signalling when conversion has concluded and new data is available for transmission. The command set for the ADS8332 is summarized below in Table 3.2, adapted from [49] — once again, only commands relevant to the proposed system are shown.

While a total of 8 input channels can be selected with different commands, the RCS only requires the use of a single input to which the photodetector is connected. There are also commands to write and

**Figure 3.4:** SPI Module finite-state machine diagram.

D[15:12]	Hex	Command	D[11:0]	R/W
0000b	0x0	Select analog input channel 0	Don't care	Write
1100b	0xC	Read CFR	Don't care	Read
1101b	0xD	Read data	Don't care	Read
1110b	0xE	Write CFR	CFR value	Write
1111b	0xF	Default mode (load CFR with default value)	Don't care	Write

Table 3.2: Summary of ADS8332 Command Set.

read the CFR, which should prove useful to validate the serial interface against the ADS8332. The “Read data” command can be used to read the conversion result: 16-bits of data in straight binary format that is output MSB-first adjusted on the serial interface. The last command, 0xF, will allow for a soft-reset of the converter (restarting system software without resetting the hardware) before any other transmissions occur, as specified previously in Section 3.1. Furthermore, electing to employ only soft resets reduces the total pinout of this module by one, which should simplify implementation and component testing efforts later into the development cycle.

After resetting the device, the CFR is set to a default value of “0xFFFF”, which does not suit the proposed design. As such, it is necessary to write the value “0x2FD” to this register, using the “Write CFR” command. This value reflects the configuration summarized below:

1. Manual channel select is enabled, by setting CFR[11]=0. The desired input channel for the internal converter is selected with the appropriate “Select channel” command;
2. The serial interface clock is programmed as the clock source of the internal conversion process, by setting CFR[10]='0';
3. The conversion process is started manually by producing a pulse on the CONVST pin — this feature is enabled by setting CFR[9]='1'.

As only a single input channel is in use, it is necessary to enable manual channel selection mode to use only the desired channel. The second and third configuration aspects highlighted have already been discussed, but have some implications that must be considered. For the configuration previously specified, on idle the converter is in a continuous state of sampling during which the data value read through the serial interface will remain the same, as the internal converter is not active. It is necessary to initiate a new conversion process, which can be done by generating a pulse on the CONVST pin. When this happens, the internal converter switches from sample to hold mode, and the conversion process is

concluded after 18 CCLKs (36 SCLKs) cycles. A minimum sampling time of 3 CCLKs (6 SCLKs) cycles must be assured before triggering new conversions. As such, the minimum time between two consecutive CONVST signals is 21 CCLKs (42 SCLKs). The conversion value is 16-bits long, requiring 16 SCLKs to collect its value through the serial interface. The converter supports two possible data readout modes: read while converting, and read while sampling. Given the previously explored configuration, reading while converting allows for the fastest conversion rates possible but comes with the caveat that the value read on a given conversion cycle N actually corresponds to the value produced during the previous conversion cycle N-1. This must be taken into account in the Laser Controller Module's design. A basic transition diagram for the FSM design is presented in Figure 3.5. Once again, each state has a set of associated Moore output signals:

1. The negative logic conversion start output signal, "o_convst";
2. The 16-bit data input signal "s_i_data", which is directly wired to the SPI Module. When serial communications are triggered, this value will be serialized out through the MOSI pin;
3. The auxiliary signal, "s_i_send", is used to trigger the SPI Module's data read/write process;
4. The auxiliary signal, "s_i_hold", is used to lock the SPI Module's state machine in the four states which implement the SCLK cycles. As such, when this signal is active, a continuous serial clock will be produced.

As before, an "Idle" state is assumed until the device is brought out of reset by an upper module. The green-coloured states of the FSM diagram correspond to the initialization process previously described: first, the device is reset via the serial interface; then the new CFR value is written; and finally, the input channel 0 is selected. Once these procedures conclude, the device enters a new idle state called "Ready". When the "i_read" input signal is active, the data readout process is executed (implemented with the red-coloured states of the FSM diagram). The state "READ 1" ensures the minimum 40ns CONVST pulse duration. State "READ 2" initiates the read-while-converting process, and state "READ 3" is held until its completion. After, state "READ 4" ensure an additional 20 SCLK cycles are produced, for a added total of 36 SCLK cycles which corresponds to the minimum 18 CCLK conversion time. Finally, state "READ 5" implements the sampling time delay of 3 CCLKs, after which the automaton returns to the "READY" state and the process may start over.

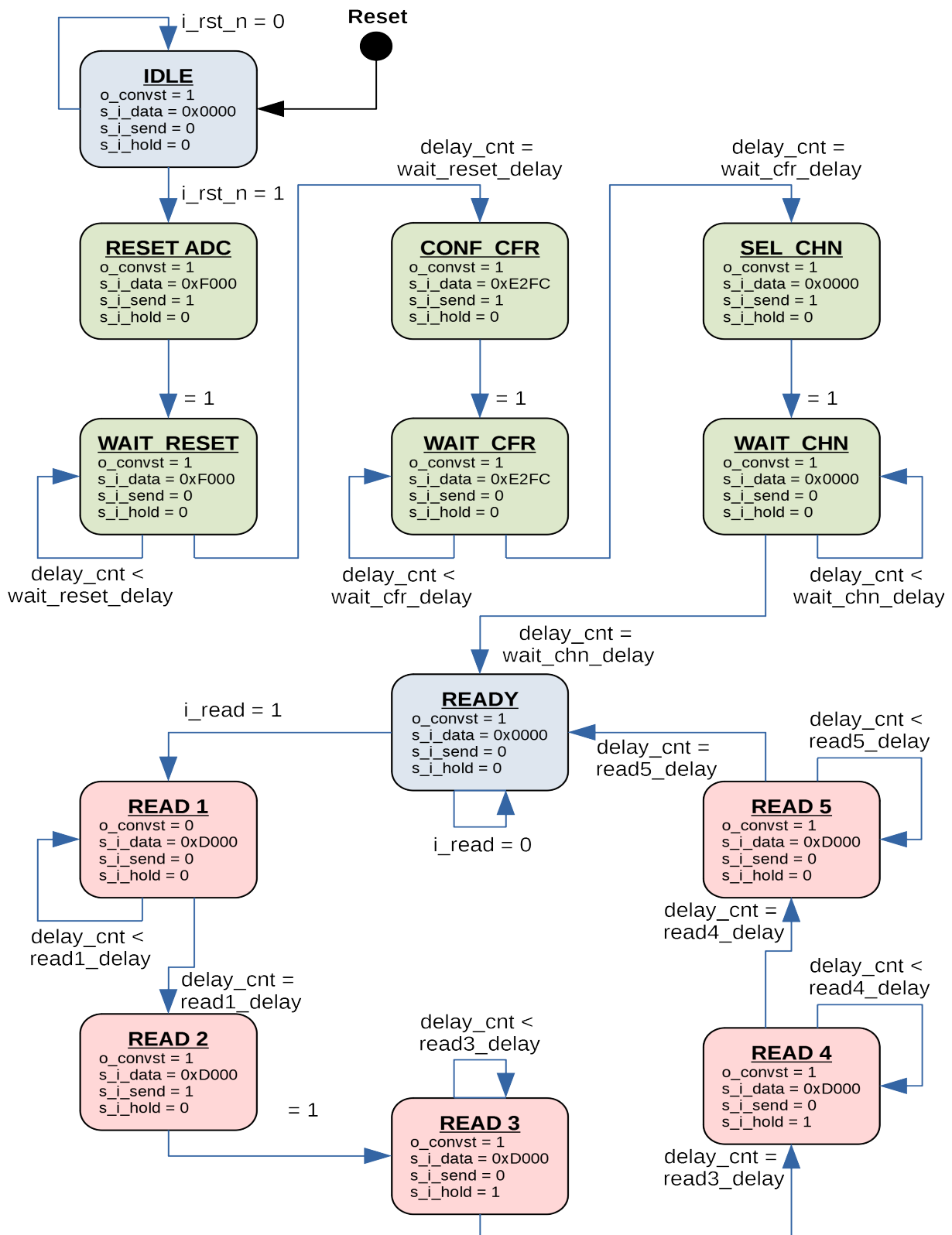


Figure 3.5: ADS8332 Controller Module finite-state machine diagram.

3.2.3 Laser Controller Module

The design of the Laser Controller Module is fairly straightforward, as it relies on the ADS8332 and SPI Modules to deal with the vast majority of the complicated timing requirements that the protocol and the converter impose. A basic transition diagram for the FSM design is presented in Figure 3.6. Before analysing, let's identify the function of each Moore output signal:

1. The auxiliary signal "s_read_pulse" is used to initiate the ADS8332 Module's data readout process, which is highlighted in red on Figure 3.5;
2. The auxiliary negative logic signal "s_rst_n" is used to keep the ADS8332 Module in its default "IDLE" state (see Figure 3.5). While it is not represented in the FSM, this signal functions as an asynchronous reset signal;
3. The auxiliary signal "s_laser_state_inc" is used to toggle the state of an additional state machine responsible for toggling each laser on and off as required, in a round-robin scheme;
4. The auxiliary signal "s_load_data" is used to store the data read from the ADC into the adequate output register.

The "RESET" state is used to keep all submodules inactive for a configurable time interval. This decision was made because after the system is powered on, the minimum time before the ADC can be reconfigured specified by the datasheet is $2\ \mu\text{s}$. For this reason, the value of "reset_delay" should ensure no serial interface activity occurs before the converter is ready. The "INIT" state is active for a configurable time interval as well, which must be long enough to allow for the ADS8332 initialization process (highlighted in green in Figure 3.5) to conclude. The "READ" state is a direct match with the data readout process of the submodule (highlighted in red in Figure 3.5). The data read from the converter is made available in external interfaces during state "EXPORT", which is followed by state "WAIT". Changing the value of "wait_delay" allows for a precise definition of the desired sample-per-second performance.

3.2.4 Test Cases

In this Section the test case for the SPI Module will be described. Every test described in this work has relied on integrating the Device Under Testing (DUT) with an AXI interface, using Vivado Design Suite's IP Packager. This utility provides a set of wizards which automate the creating and packaging of a custom IP, and made it easy to instantiate AXI ports and signals. This meant the final module could be connected

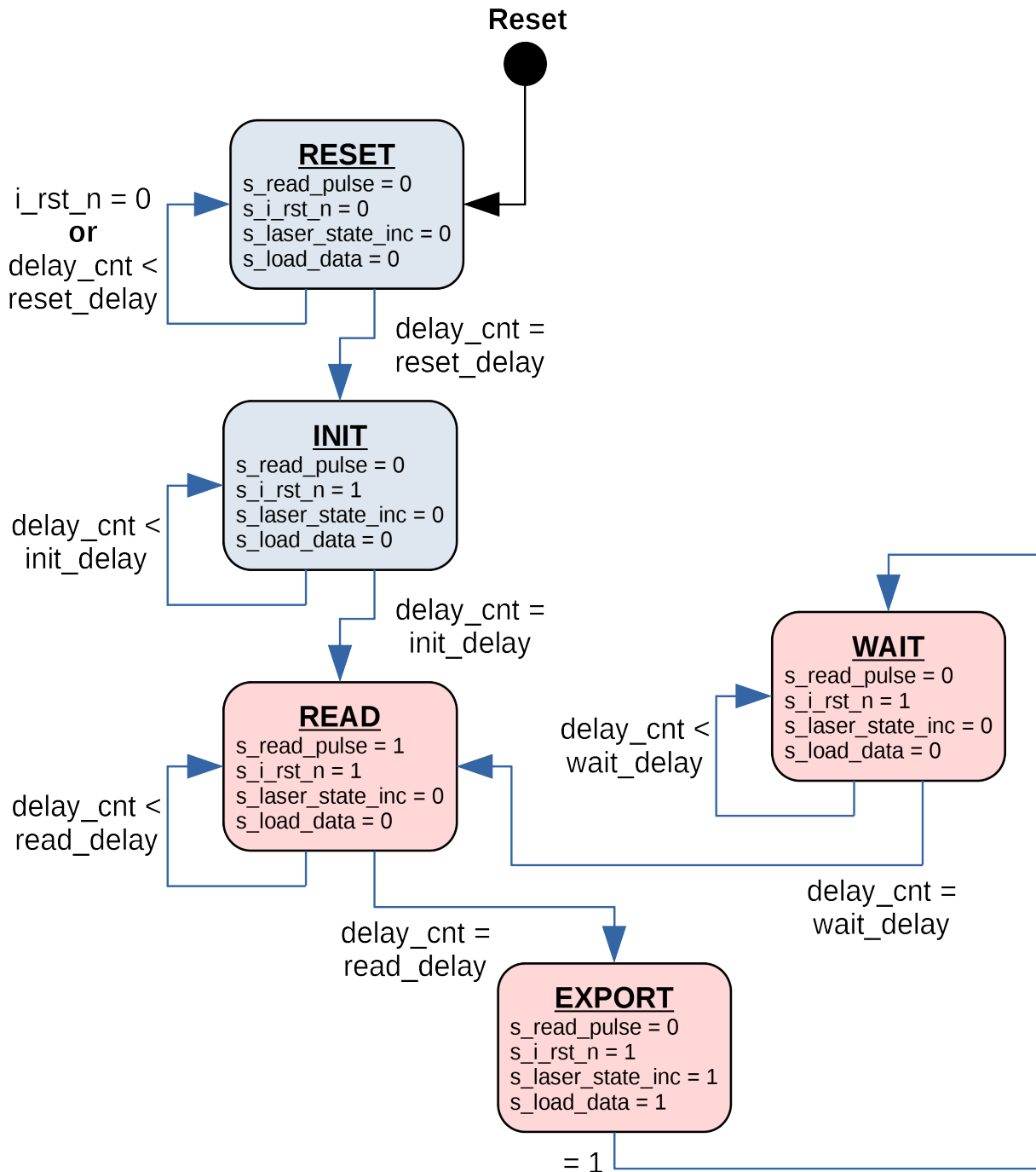


Figure 3.6: Laser Controller Module finite-state machine diagram.

to either the Zynq-7000 PS or a soft-core instantiated in the PL, which allowed for a new level of flexibility and control of the testing procedures: tests could now benefit from the step-through debugger, breakpoint functionalities and memory monitor provided by the Vitis IDE. The standard Pmod JE is used to connect the SPI protocol signals from the ZYBO Z7-10 to the Arduino Uno REV3 board, as shown below in Table 3.3. The Arduino board must be programmed with a simple echo application where it operates as an SPI Slave.

ZYBO Z7-10		Arduino Uno REV3	
Pin	Signal	Pin	Signal
J15	o_mosi	D11	MOSI
V12	o_cs_n	D10	SS
W16	i_miso	D12	MISO
H15	o_sclk	D13	SCLK

Table 3.3: Input/output connections between ZYBO Z7-10 and Arduino Uno REV3 for SPI Module testing.

Furthermore, the 3.3 V logic level must be selected with the appropriate jumper, as that is the maximum voltage at which the ZYBO's input/output pins operate. A software application must also be developed for the ZYBO, in which SPI messages with distinct contents are sent in a loop. The expect result is that when transmission N occurs, the Arduino should reply with the contents of the previous message N-1.

3.3 Implementation & Evaluation

3.3.1 SPI Module

The implemented SPI Module is a close match to what was described previously in Section 3.2.1. The interface of the final component is presented below in Listing 1.

The final module implements some added configurability options, as evidenced by the “WORD_SIZE” and “CLK_PRESC” interface constants — known as generic values in VHDL language nomenclature. The first allows for a configurable transfer length, allow for quick adaptation of the module so it can also be used to establish communications with other SPI capable devices. The “CLK_PRESC” generic is used to implement a simple clock divider that takes an input signal of a frequency, f_{i_clk} , and generates an output

```

1  entity spi_driver is
2      generic (
3          WORD_SIZE : positive := 16;    -- data bus width / transfer length
4          CLK_PRESC : integer := 0      -- i_clk is divided by CLK_PRESC+1
5      );
6      port (
7          -- interface with upper module
8          i_data  : in std_logic_vector (WORD_SIZE-1 downto 0);
9          o_data  : out std_logic_vector (WORD_SIZE-1 downto 0);
10         i_send  : in std_logic;
11         i_hold  : in std_logic;
12         -- spi interface
13         o_sclk  : out std_logic;
14         o_mosi  : out std_logic;
15         i_miso  : in std_logic;
16         o_cs_n  : out std_logic;
17         -- control signals
18         i_clk   : in std_logic;
19         i_rst_n : in std_logic
20     );
21 end spi_driver;

```

Listing 1: SPI Module's VHDL interface.

signal with a frequency defined by Equation 3.2:

$$f_{i_clk_new} = \frac{f_{i_clk}}{CLK_PRESC + 1} \quad (3.2)$$

This capability was used for the test case described previously in Section 3.2.4. It was necessary to test the device at lower speeds (without lowering the clock frequency of the AXI4 wrapper interface) for waveform validation using an oscilloscope, as the inductance of the testing equipment limited the signal switching speeds at which discernible clock waveforms could be observed. The tests were performed with the clock frequency given in Equation 3.3 below:

$$f_{i_clk_new} = \frac{f_{i_clk}}{CLK_PRESC + 1} = \frac{50 \text{ MHz}}{30 + 1} = 166 \text{ kHz} \quad (3.3)$$

The final serial interface baudrate is given by Equation 3.4:

$$f_{SCLK} = \frac{f_{i_clk_new}}{4} = \frac{166 \text{ kHz}}{4} = 41.5 \text{ kHz} \quad (3.4)$$

As previously described in Section 3.2.4, Vivado's IP Packager was used to create a custom IP containing an AXI interface with 4 internal 32-bit registers was generated. The implemented SPI core was integrated into this IP, with each signal of its interface now connected to the aforementioned registers

which enabled adequate read/write functionalities depending on the type of port. The final test system includes a Zynq-7000 processing system, with an enabled Master AXI interface connected with the AXI SPI Module via an AXI Interconnect and Processor System Reset IP, as shown below in Figure 3.7:

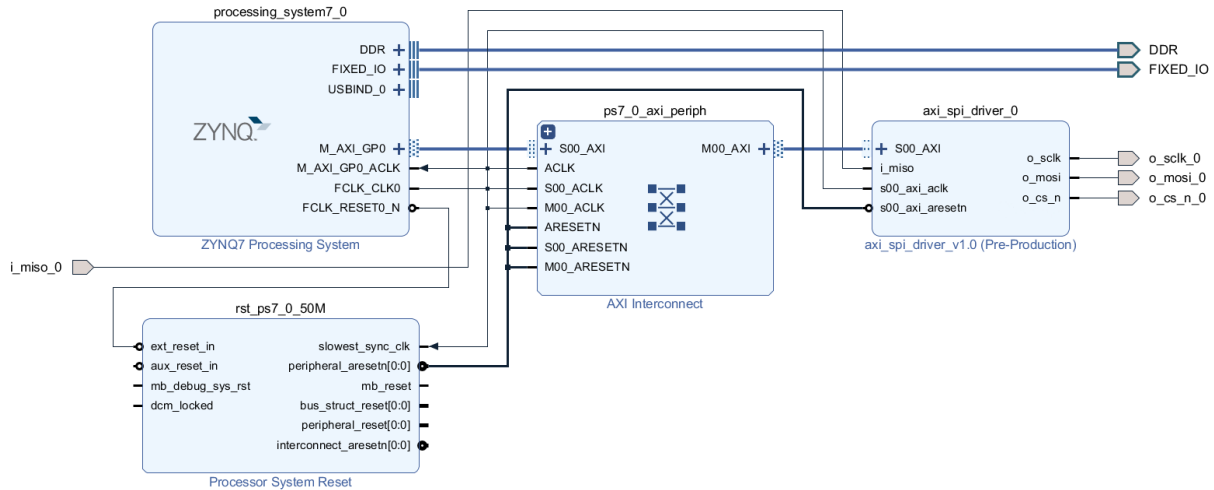


Figure 3.7: Block Design View of SPI Module AXI test.

In this setup, the AXI SPI Module can be easily controlled as a memory-mapped device, as shown below in Listing 2 which presents the simple test application developed in Vitis IDE.

```

1  void spi_arduino_test ()
2  {
3      volatile u32 spiPtr = XPAR_AXI_SPI_DRIVER_0_S00_AXI_BASEADDR;
4      volatile u32 tempData = 0;
5      while (1)
6      {
7          sleep(1);
8          Xil_Out32(spiPtr+4*2, 0x00);           // i_send = '0'
9          Xil_Out32(spiPtr, 0x1234);
10         Xil_Out32(spiPtr+4*2, 0x01);           // i_send = '1'
11         tempData = Xil_In32(spiPtr+4*1);       // Value read from Arduino
12
13         sleep(1);
14         (spiPtr+4*2, 0x00);                     // i_send = '0'
15         Xil_Out32(spiPtr, 0x5678);
16         Xil_Out32(spiPtr+4*2, 0x01);           // i_send = '1'
17         tempData = Xil_In32(spiPtr+4*1);       // Value read from Arduino
18     }
19 }

```

Listing 2: Detail of the SPI Module's C++ testing application deployed on the ZYBO Z7-10's PS.

The Arduino was loaded with a simple echo application, where it played the role of an SPI Slave (configured with CPOL=1 and CPHA=0) that remains idle until the “chip select” line is forced low by the Master. Message reception is handled by an Interrupt Service Routine (ISR), which stored messages in

a reception buffer that was also used as the transmission buffer — thus, it “echoed” received messages. The current contents of this internal buffer were printed on a UART interface when the user button was pressed, to facilitate debugging efforts. The generated waveform, as captured using a digital oscilloscope, is shown below in Figure 3.8.

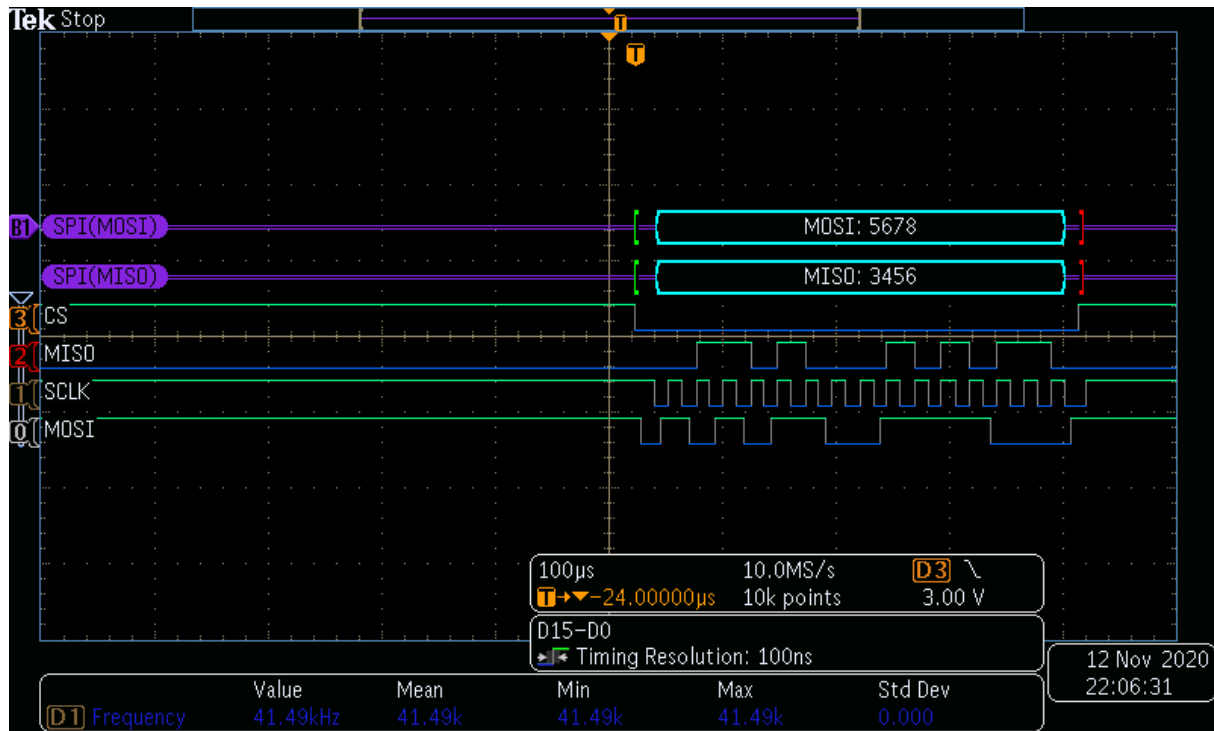


Figure 3.8: SPI Module's test case results.

The waveform shown was captured when the second message was sent from the SPI Module in the ZYBO. At first, it may seem strange that the Arduino transmits “0x3456” on its serial output line, when in reality this is due to some of the Arduino’s configuration limitations. When in Slave mode, the Arduino’s SPI is hardwired to receive and transmit 8-bits of data, while for this test the Zybo was configured to send a message that is twice as long (16-bits). As such, when the second SPI message containing “0x5678” is sent by the Master device, the Arduino responds first with an 8-bit message containing its current internal buffer’s content (which match the last 8-bits of the previous “0x1234” message). Then, as the ZYBO continues transmitting the final 8-bits of the message, the Arduino’s internal buffer contents are now “0x56”, which are promptly serialized. The results obtained prove the SPI Module is working correctly. Furthermore, the frequency measurements facilitated by the digital oscilloscope show that the SCLK frequency matches its expect value of 41.5 KHz, validating the clock divider’s functionality.

The total configurable hardware resources consumed by the stand-alone SPI Module (without the AXI interface) are presented below in Table 3.4.

Resource	Utilization	Available	Utilization %
LUT	28	17600	0.1590909
FF	58	35200	0.16477272
IO	40	100	40.0
BUFG	1	32	3.125

Table 3.4: SPI Module resource utilization (WORD_SIZE=16).

The total resource usage is extremely low, as would be expected for such a simple module. The higher IO usage percentage may seem problematic, but when this module is integrated into the final system its ports won't be consuming these resources as they will be routed internally with other configurable hardware modules. The SPI module achieves a maximum operating clock frequency of 333 MHz, at which the serial interface clock generated in operation would be 83.3 MHz. For interfacing with the ADS8332 at 20 MHz an input clock of 80 MHz is required, which is evidently possible with the proposed design.

To evaluate this module's suitability for interfacing with the ADT7301 and DACC8162, it was necessary to discover the maximum serial clock frequency at which the module could be run while still respect each device's timing requirements. This process was only verified using Vivado's simulator, using the waveform window to manually verify whether timing specifications were met or not. A summary of this device's timing requirements, as well as the SPI module's timing specifications are summarized in Table 3.5, adapted from [50].

Parameter	Limit	Unit	SPI Module (ZYBO)
CS to SCLK setup time	5	ns min	60 ns
SCLK high pulse width	25	ns min	40 ns
SCLK low pulse width	25	ns min	40 ns
Data setup time prior to SCLK rising edge	20	ns min	20 ns
Data hold time after SCLK rising edge	5	ns min	60 ns
CS to SCLK hold time	5	ns min	40 ns

Table 3.5: Summary of ADT7301 serial interface timing requirements.

Read operations occur during streams of 16 clock pulses, so the SPI Module was configured with a transfer length of 16-bits. As evidenced in the timing summaries, the simulated maximum SPI clock frequency suitable for interfacing the SPI Module with the ADT7301 is 12.5 MHz. This value can be

achieved with an input clock frequency of 50 MHz and without any clock division ($\text{CLK_PRESC}='1'$), as shown in Equation 3.5:

$$f_{i_clk} = f_{SCLK} \times 4 = 12.5 \text{ MHz} \times 4 = 50 \text{ MHz} \quad (3.5)$$

The digital temperature sensor performs internal data conversion once every 1.5 seconds, which makes sampling rates higher than 0.66 SPS unnecessary and redundant, as every other data value read from the sensor would be a duplicate. In the configuration proposed, the SPI Module can perform data readouts in 1300 ns (measured from falling- to rising-edge of CS), making it quite suitable for interfacing with the ADT7301 temperature sensor. The same timing analysis was conducted for the DAC8162, the results of which are summarized below in Table 3.6, adapted from [48].

Parameter	Limit	Unit	SPI Module (ZYBO)
SCLK frequency	50	MHz max	aprox. 41.6 MHz
SCLK high pulse width	8	ns min	12 ns
SCLK low pulse width	8	ns min	12 ns
Data setup time prior to SCLK rising edge	6	ns min	6 ns
Data hold time after SCLK rising edge	5	ns min	6 ns

Table 3.6: Summary of DAC8162 serial interface timing requirements.

It is important to note here that the DAC8162's digital interface features other signals with associated timing requirements that have not been considered for this analysis, as the SPI Module is concerned only with implementing the serial protocol. That said, the analysis suggests a maximum SPI clock frequency of around 41.6 MHz, which can be derived directly from an input clock of 166.6 MHz.

3.3.2 ADS8332 Controller Module

The interface of the final component is presented below in Listing 3. There are no substantial differences from what was defined in the design stage (Section 3.2.2).

The delays between FSM state transitions were implemented with an auxiliary timer, and can be adjusted manually as long as the minimum delay values are respected (projected for a serial clock frequency of 20 MHz). The “wait_reset_delay” must be no lower than 875 ns, which correspond to the necessary 16 SCLKs for the 16-bit message transmission, with an added 75 ns of padding to perform state transitions. The same lower limit holds true for “wait_cfr_delay” and “wait_chn_delay”. The “read1_delay” is 50 ns

```

1  entity ads8332_driver is
2      port(
3          -- interface with upper module
4          i_read      : in std_logic;
5          o_data       : out std_logic_vector(15 downto 0);
6          -- spi Interface
7          o_convst     : out std_logic;
8          o_sclk       : out std_logic;
9          o_mosi       : out std_logic;
10         i_miso       : in std_logic;
11         o_cs_n       : out std_logic;
12         -- control signals.
13         i_clk        : in std_logic;
14         i_rst_n      : in std_logic
15     );
16 end ads8332_driver;

```

Listing 3: ADS8332 Controller Module's VHDL interface.

or higher, as it is responsible for the active low logic pulse on the CONVST pin. The “read3_delay” shares the 875 ns lower boundary of previously described delays, while “read4_delay” has a minimum duration of 1000 ns. The final configurable delay “read5_delay” must equal or exceed 300 ns in duration, as it implements the minimum 3 CCLKs of sampling time required by the converter. The ADS8332 Controller module achieves the same maximum operating clock frequency of 333 MHz of the SPI Module, which again vastly exceeds the target operational frequency of 80 MHz that produces the desired 20 MHz serial clock frequency. In these conditions, the ADS8332 initialization procedures take a total of 2668 ns to conclude, measured from the release of the module’s reset signal until “STATE_READY” is reached — this process is illustrated in Figure 3.9, as seen in Vivado Simulator..

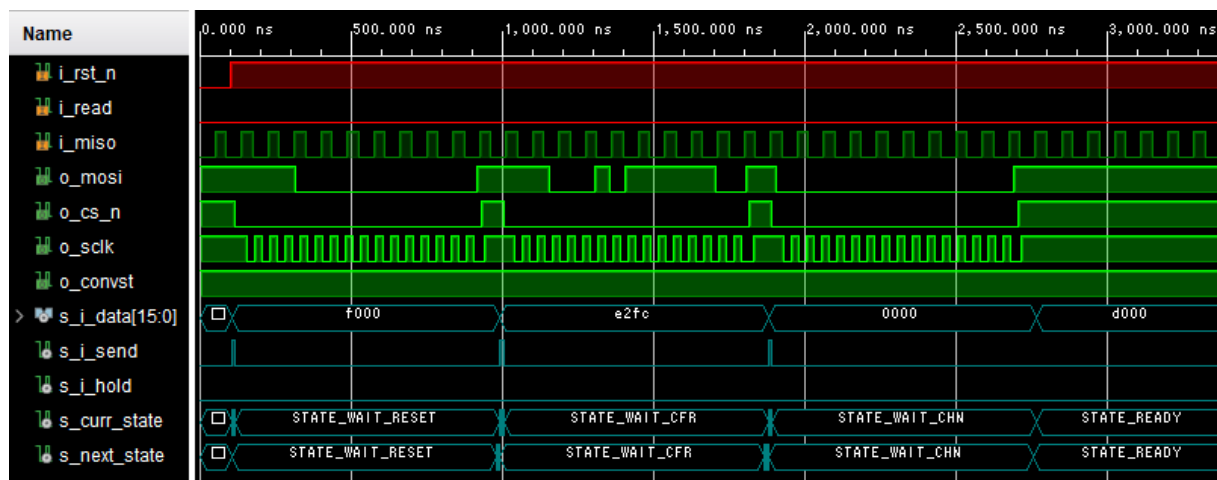


Figure 3.9: Detail of ADS8332 Controller's initialization process

Using the same setup, and assuming constant data readout requests from an upper module (simulated

by holding the “i_read” signal active), a time interval of 2175 ns can be measured between falling edges of the CONVST pin. This corresponds to a maximum sampling rate of approximately 460 kSPS, which is just under the maximum sampling rate of 500 kSPS that the ADS8332 supports in Auto-trigger mode. The resource usage of this module is presented below in Table 3.7. The resource utilization is slightly higher than the one observed for the SPI Module (see Table 3.4), but once again not concerning.

Resource	Utilization	Available	Utilization %
LUT	91	17600	0.5170455
FF	95	35200	0.26988637
IO	24	100	24.0
BUFG	1	32	3.125

Table 3.7: ADS8332 Controller Module resource utilization.

3.3.3 Laser Controller Module

The interface of the final component is presented below in Listing 4, and includes some additional signals that must be analysed. The “o_laser” output signals implement the interface with the laser diodes, which will be toggled in a round-robin scheme. The module interface also includes four 16-bit wide data outputs and a “o_load” signal. Each one of the four laser having its own associated “o_dataN” output bus, and once one round of data readouts is perform the “o_load” signal becomes active to advertise that the newly acquired data samples are ready for processing.

The implementation of delays for FSM state transitions followed the same strategy employed for the ADS8332 Controller Module, allowing for great granularity when specifying the desired data acquisition timing specifications as long as the minimum delay values are respected (projected for a serial clock frequency of 20 MHz). The “reset_delay” was set at a recommended 2 μ s for reasons previously exposed in Section 3.2.3 — however, this interval depends on the final design’s power-on procedures and it may be possible to shorten it. The “init_delay” length can be set at a minimum value of 2825 ns, corresponding to the sum of initialization delays of the ADS8332 Controller module with an added 200 ns of padding to account for state transitions. The “read_delay” interval can be as short as 2250 ns, which slightly exceeds the 2175 ns minimum time interval measured between falling edge of the CONVST pin permitted by the ADS8332 Controller Module. This has the effect of lowering the maximum sample rate to 444 kSPS, which is 11% lower than the advertised 500 kSPS maximum sample rate of the ADC. The duration

```

1  entity laser_controller is
2      port(
3          -- Control Signals
4          i_clk      : in std_logic;
5          i_rst_n    : in std_logic;
6          -- Outputs
7          o_laser     : out std_logic_vector(3 downto 0);
8          -- Interface with laser2sum IP
9          o_data0     : out std_logic_vector(15 downto 0);
10         o_data1     : out std_logic_vector(15 downto 0);
11         o_data2     : out std_logic_vector(15 downto 0);
12         o_data3     : out std_logic_vector(15 downto 0);
13         o_load      : out std_logic;
14         -- ADS8332 Interface
15         o_sclk      : out std_logic;
16         o_mosi      : out std_logic;
17         i_miso      : in std_logic;
18         o_cs_n      : out std_logic;
19         o_convst     : out std_logic;
20     );
21 end laser_driver;

```

Listing 4: Laser Controller Module's VHDL interface.

of “wait_delay” has no imposed lower or higher bound, as it is provided merely for added design flexibility.

The total FPGA resource utilization of the Laser Controller Module is presented below in Table 3.8.

Resource	Utilization	Available	Utilization %
LUT	141	17600	0.8011364
FF	150	35200	0.42613637
IO	28	100	28.0
BUFG	1	32	3.125

Table 3.8: Laser Controller Module resource utilization.

Once again, the choice to implement a custom IP (rather than using a generic AXI-compatible Xilinx IP) is rewarded by extremely low resource utilization. The maximum clock frequency possible is now 200 MHz, which again eclipses the targeted 80 MHz which produce the desired serial clock frequency.

3.4 Discussion

The proposed modules show the viability as well as the several benefits and drawbacks when implementing the RCS’s data acquisition and laser control algorithms in reconfigurable hardware blocks: the

benefits in real-time performance and low resource usage are counteracted by a much higher development time and lower flexibility. The proposed system could be improved by enabling some of the ADS8332's advanced features which were unexplored for this work. The sensor is capable of entering low energy modes when not in use, swiftly returning to activity and providing converted values upon request. Another feature that may be of interest is its auto-trigger capabilities, which allow for the highest possible sample rates (500 kSPS). However, it is possible that these features do not fit the precise data acquisition mechanisms necessary for the RCS. Another important step which was omitted for this work is the validation of the two upper modules of the prescribed hierarchy: the ADS8332 Controller and the Laser Controller Modules.

Chapter 4

LASER2SVM Module

The data periodically collected by the modules described in the previous Chapter must undergo a pre-processing stage, before being ready for the inference process of the selected SVM algorithm. More specifically, a total of six SVM input values must be produced, which correspond to ratios between the four ADC data samples collected most recently retrieved. Furthermore, the desire to operate the data retrieval and the SVM inference modules at distinct clock frequencies requires the implementation of an adequate mechanism to transport data values between the two clock domains.

This Chapter describes the LASER2SVM Module that handles the aforementioned functionality. Section 4.1 provides a brief analysis on the challenges of performing division in hardware and crossing clock domains, which justify the need for this application-specific module. Section 4.2 include a complete description of the design process, while Section 4.3 explores the final implementation of the module. Finally, Section 4.4 highlights the future testing and validation work that must be conducted before including this module on the final RCS system.

4.1 Analysis

Performing division requires more thought and consideration from the hardware designer than the other four basic arithmetic operations (adding, subtracting and multiplying). Two rational numbers may be multiplied, added or subtracted with absolute accuracy whenever sufficient bit length is allowed for the result. This enables synthesis tools to produce accurate circuits from these operators in the source code, given that the correct operand and result lengths are specified. For example, multiplying a 16-bit value by an 8-bit value always produces a result that can be accurately represented with 24-bits. Unfortunately, this is not shared with division, as the bits necessary to store the result cannot simply be derived from the length of the operands. An obvious example of this is the division of 1 by 3 which results in an irrational number whose decimal representation is infinitely long (i.e. 0.33333...). Since there is no finite number of bits that

can represent all the fractional bits of this number, the length of the result is a function of the application's requirements and how much error it can tolerate. Increasing the length of the result improves accuracy, but requires more hardware resources and computation cycles to produce the quotient. In sum, the design of division algorithms and implementations is a game of trade-offs. Consequently, most hardware synthesis tools will not synthesize dividers. Xilinx's UG901 [56] details that both VHDL and Verilog division expressions are only supported for synthesis if the divisor is a constant power of 2 (replacing the division by a bitwise shift), or if both operands are constant. This does not occur in the RCS, since both operands come from the embedded sensor and their value may change with each new data readout. As such, it is necessary to implement a module that receives the data samples from the data collection mechanisms previously described in Chapter 3), and produces the adequate inputs for the subsequent SVM inference stage. The two main functional requirements for this module are:

1. Perform division on four 16-bit binary operands upon request;
2. Implement a Clock Domain Crossing (CDC) [57] mechanism to transfer the ADC data samples, which were produced inside the data acquisition module's clock domain, to the new data processing clock domain.

An additional non-functional requirements was identified, which would greatly facilitate design testing and iteration: designing a module with parametrizable input and output widths. As the operand width of the fixed-point datapath may change during the iteration process, an easily adjustable connection module between the data acquisition and the classification algorithm is desired.

4.2 Design

The proposed division algorithm uses fixed-point arithmetic, instead of floating-point formats widely adopted in generic purpose computing solutions. The rationale behind this design choice is extensively explored in Section 5.2, as the success discovered while implementing the SVM fixed-point architecture has ultimately enabled the design of a division algorithm in the same notation.

The most widely-adopted division algorithm can be seen as belonging to one of two classes: digit recurrence algorithms and functional iteration algorithms. The former produce a single bit of the quotient at every computation step, such that the computational complexity, i.e. the amount of computer time it takes to run an algorithm, scales linearly with the number of bits of the quotient — the traditional paper and pencil division technique taught in elementary school belongs to this class. The latter class of algorithms establish

an initial estimation, and then employ function-solving techniques to converge towards the final quotient and remainder, with the specified precision. The computational complexity of these algorithms is better than what is possible with digit recurrence algorithms, but multipliers are required. Dedicated hardware multipliers are one of the scarcest FPGA resources, and implementing them with generic configurable logic blocks is resource-expensive as well.

The selected algorithm for this module belongs to the class of digit recurrence algorithms, usually selected due to their lower complexity and resource usage when working with signed two's-complement binary numbers. It is a non-recurring division algorithm [58], based on a common digit recurrence method.

$$B^P \times X = Q \times Y + R, \quad (4.1)$$

In Equation 4.1: B is the base, or radix, used to represent the numbers; P is the number of fractional bits in the quotient; X is the integer dividend; Q is the quotient; Y is the natural dividend; and R is the remainder. This basic algorithm applies only when the dividend, Y, is greater than the divisor X. To understand how this method is applied, it is best to consider the non-recurring division algorithm presented in Listing 5 below.

Listing 5 highlights three different stages to this algorithm: preliminary operations, main recursive loop and terminal operations. The preliminary operations involve loading the remainder with the value of the dividend ($R(0) := X$). The main recursive loop employs a recursive index that goes from the first bit 0 to the last bit P-1 of the unknown quotient. The values of $Q(i)$ and $R(i+1)$ are defined with each new iteration based on the value of $r(i)$, the partial remainder calculated in the previous step. If $r(i) < 0$ then $Q(i) := 0$, and the previous partial remainder $R(i)$ is left-shifted (multiplied by the radix B) and Y is added to produce $R(i+1)$. If not, then $Q(i) := 1$, and the previous partial remainder $R(i)$ is left-shifted (multiplied by the radix B) and Y is instead subtracted to produce $R(i+1)$. The following step ensures the quotient is converted from a B's complement representation back to a signed representation. The final step ensures that the remainder and the quotient have the same sign. For further clarity, an example is provided in Listing 6, representing the division of -12 by 5 with an accuracy of 8 fractional bits.

Besides performing division, this module is tasked with implementing a CDC mechanism. The data collection solution described in Chapter 3 operates at a target input clock frequency of 80 MHz. However, the subsequent SVM classifier would benefit greatly if it could be instantiated with the highest clock frequencies possible, to ensure the most time-consuming task in the whole RCS data computation process is executed as swiftly as possible. This introduces the problem of having two different clock domains in the system, which can lead to metastability issues if left unattended.

```

1  // Preliminary operations
2  r[0]=X;
3
4  // Main recursive loop
5  for (i = 0; i <= P-1; i++ ) {
6      if (r[i]<0) {
7          Q[i]=0;
8          R[i+1]=B*R[i]+Y;
9      } else {
10         Q[i]=1;
11         R[i+1]=B*R[i]-Y;
12     }
13 }
14
15 // Conversion to signed representation
16 Q[0]=1-Q[0];
17 Q[P]=1;
18 R=R[P];
19
20 // Terminal operations
21 if (X>=0 && R<0) {
22     R=R+Y;
23     Q=Q-1;
24 } else if (X<0 && R>=0) {
25     R=R-Y;
26     Q=Q+1;
27 }

```

Listing 5: Non-restoring division's main computation loop.

A clock domain in a digital circuit is defined as the set of logic elements that function on the same clock event, i.e. the same clock edge of the same clock signal. That is to say, if sets of logic elements depend of different clock signals, or indeed a different edge of the same signal, it is said that those sets belong to independent clock domains. Elements (both sequential and combinational) belonging to a single clock domain form a circuit free of metastability, facilitated by timing analysis and synthesis tools.

Every sequential logic element — the most basic being the Flip-Flop (FF) — defines strict setup and hold times, during which the data input must be stable before and after the clock event. Transferring signals between asynchronous clock domains may lead to setup or hold timing violations, causing the output of the FF to oscillate for a indefinite amount of time where its logic level is neither '0' or '1', a condition known as metastability. This undefined value may not converge before a subsequent sequential logic element samples its value, propagating the metastable condition throughout the system, causing unpredictable behaviour. A synchronizer circuit must be implemented to ensure data is safely transferred between clock domains. There are number of techniques, the most common being a 2- or 3-FFs chain where each FF takes the clock from the new clock domain to capture the signal generated in the old clock domain. This

```

1  // Preliminary operations:
2  r[0] = -12 < 0,
3  // Main recursive loop:
4  q[0] = 0, r[1] = -24 + 15 = -9 < 0,
5  q[1] = 0, r[2] = -18 + 15 = -3 < 0,
6  q[2] = 0, r[3] = -6 + 15 = 9 > 0,
7  q[3] = 1, r[4] = 18 - 15 = 3 > 0,
8  q[4] = 1, r[5] = 6 - 15 = -9 < 0,
9  q[5] = 0, r[6] = -18 + 15 = -3 < 0,
10 q[6] = 0, r[7] = -6 + 15 = 9 < 0,
11 q[7] = 1, r[8] = 18 - 15 = 3 < 0
12 q[8] = 1
13 // After main recursive loop, values are in 2's complement format:
14 Q = 000110011
15 R = R[8] = 3
16 // After conversion to signed representation:
17 Q = 1001100112 = -20510
18 R = 310
19 // After sign correction:
20 Q = Q+1 = -205+1 = -20410
21 R = R-Y = 3-15 = -1210
22 // Considering p=8, results should be interpreted as:
23 Q = -204 ÷ 28 = -0.79687510
24 R = -12 × 2-8 = -0.04687510

```

Listing 6: Example of non-recurring division of -12 by 15, with P=8.

technique does not eliminate, but rather substantially mitigate the possibility of metastability at the output of the last sequential element in the chain.

However, given that the Laser Controller is a completely synchronous design with well-defined timing operation characteristics and that the SVM Classifier will surely function at a higher clock frequency, it is possible to exploit these qualities to produce a more sophisticated synchronizer that transfers a group of signals at once, with the cost of some added latency. The proposed solution involves a hand-shaking mechanism, and is illustrated in Figure 4.1, where it is used to transfer 8-bits of data between two clock domains.

The example provided illustrates a clock domain “A” that wishes to transfer data to an independent clock domain “B”. Once the sender has new data to transmit, it must place it on the bus and activate the “REQUEST” signal. This signal is collected using a chain of 2-FFs on the new clock domain which produce the “NEW_REQ” signal. When this signal is active, the receiver will then clock the data available on the bus into its clock domain and answer with the “ACKNOWLEDGE” signal. The identical 2-FFs chain approach is used to produce the “NEW_ACK” signal on the sender’s clock domain. The sender may then deactivate its “REQUEST” signal, which will prompt the “NEW_ACK” to return to an inactive state after a certain period of time. The sender is said to be “busy” while the “REQUEST” is active or the “NEW_ACK”

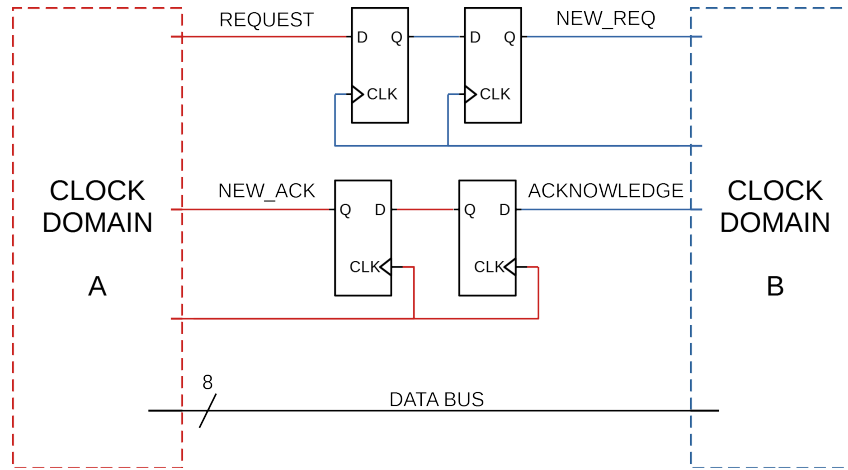


Figure 4.1: Illustration of selected CDC Handshake mechanism.

signal is active, and during this period it cannot change the value being driven onto the data bus or issue new transfer requests.

4.3 Implementation & Evaluation

The VHDL interface of the proposed LASER2SVM component is presented below in Figure 7. It features two clock inputs: one coming from the Laser Driver IP, the other coming from the SVM IP. Four 16-bit data inputs (each corresponding to a sample captured while a specific laser diode was active) and a “i_load” signal complete the interface with the Laser Controller module. The interface with the SVM IP is composed of six 18-bit wide data outputs and a “o_load_2” signal. The “i_load_1” is active for a single “i_clk_1” clock cycle when the Laser Controller IP has produced four new ADC samples. The “o_load_2” signal is active for one “i_clk_2” clock cycle when the LASER2SVM module has finished computing the six data values that feed into the SVM Classifier.

This module instantiates six internal dividers which operate in parallel to produce the six 18-bit data outputs. The division submodules’ implementation was made to enable quick design iteration, as can be seen below in Listing 8.

The dividers’ input and output widths are completely parametrizable, which enables faster iterations as future SVM designs (or even other types of classifiers, for that matter) may require different datapath widths. At first glance, there seems to be a mismatch between the specified 18-bit data outputs of the LASER2SVM module (as shown in Listing 7) and the “OUTPUT_SIZE” parameter set to 25, as shown in Listing 8. This relates to the fact that the SVM IP specifies 18-bit inputs in the S_8_9 format (1 sign, 8 integer and 9 fractional bits). The dividers accept 16-bit signed integer values and produce 25-bit signed fixed-point

```

1  entity laser2svm is
2      port (
3          -- Control Signals
4          i_clk_1      : in std_logic;      -- Laser Driver IP CLOCK
5          i_clk_2      : in std_logic;      -- SVM IP Clock
6          -- Interface with Laser Controller IP
7          i_data0_1    : in std_logic_vector (15 downto 0);  -- 980nm
8          i_data1_1    : in std_logic_vector (15 downto 0);  -- 1310nm
9          i_data2_1    : in std_logic_vector (15 downto 0);  -- 1410nm
10         i_data3_1    : in std_logic_vector (15 downto 0);  -- 1550nm
11         i_load_1     : in std_logic;
12         -- Interface with SVM IP
13         o_data0_2    : out std_logic_vector (17 downto 0);
14         o_data1_2    : out std_logic_vector (17 downto 0);
15         o_data2_2    : out std_logic_vector (17 downto 0);
16         o_data3_2    : out std_logic_vector (17 downto 0);
17         o_data4_2    : out std_logic_vector (17 downto 0);
18         o_data5_2    : out std_logic_vector (17 downto 0);
19         o_load_2     : out std_logic
20     );
21 end laser2svm;

```

Listing 7: LASER2SVM Module's VHDL interface.

quotients in the S_{15_9} format. The results go through a saturation stage ensuring their magnitude does not exceed 255 to produce the final division results in the desired S_{8_9} format. The dividers incorporate an extra initial alignment procedure where the 16-bit divisor input, Y , is internally replaced with Y' such that $Y' = B^{15} \times Y$. The dividend input, X , is similarly replaced by $X' = X/2$. These transformations ensure that the $-Y \leq X < Y$ requirement is met. The divider module implements a simple FSM that governs its behaviour. Upon reset of the system, the module enters the “IDLE” state in which it awaits the activation of the “i_start” signal. Once that occurs, it begins the division operations by transitioning to the “LOOP” state. It is in this state that the main recursive loop is implemented (see lines 4 to 12 of Listing 6), taking “OUTPUT_SIZE” clock cycles to conclude. After this stage, it enters the “FIX” state where the quotient and remainder values undergo the sign correction procedure to produce the final result, after which the module returns to the “IDLE” state. The CDC Handshake VHDL implementation is shown in Listing 9. The request signal from clock domain “A” (see Figure 4.1) here takes the form of the “i_load_1” signal, with the acknowledge signal being implemented as the “s_last_red” signal. The additional “reqpipe_proc” process (lines 17 to 24) guarantees that only a 0-to-1 transition of the request signal causes the data bus' values to be internally loaded. When this occurs the “s_start” signal is also toggled on (line 42), marking the beginning of the parallel division operations. This implementation exploits the fact that the event that activates the request signal is a “rare” event — more specifically, it is when four new ADC samples are available at the outputs of the Laser Controller module. This event occurs at a known lower frequency

```

1  entity non_restoring is
2      generic(
3          INPUT_SIZE  : natural := 16;
4          OUTPUT_SIZE : natural := 25
5      );
6      port(
7          i_clk       : in  std_logic;
8          i_rst       : in  std_logic;
9          i_start      : in  std_logic;
10         i_dividend   : in  std_logic_vector(INPUT_SIZE-1 downto 0);
11         i_divisor    : in  std_logic_vector(INPUT_SIZE-1 downto 0);
12         o_quotient   : out std_logic_vector(OUTPUT_SIZE-1 downto 0);
13         o_remainder  : out std_logic_vector(OUTPUT_SIZE-1 downto 0);
14         o_done       : out std_logic
15     );
16 end non_restoring;

```

Listing 8: Non-restoring Division Module's VHDL interface.

such that the CDC circuit is never busy handling the last event when the new event occurs. This means that the data outputs are assuredly stable during the “busy” period of the CDC mechanism, even without explicitly providing this signal as an input to the Laser Controller module that drives the data values to be transferred.

The post-synthesis resource estimation is shown below in Table 4.1. Stand-alone implementation of this module is not possible as it requires a vastly superior number of I/O resources, as the Table shows. For this reason there are no stand-alone maximum frequency or validation tests provided: these are instead shown in Section 5.3.2 where it is integrated into a whole RCS system design. It is the most resource consuming module presented so far, with an estimated 353 LUTs and 587 FFs utilized.

Resource	Utilization	Available	Utilization %
LUT	353	17600	2.0056818
FF	587	35200	1.6676137
IO	176	100	176.0
BUFG	2	32	6.25

Table 4.1: LASER2SVM Module resource utilization.

Sutter et al. [59] presented a traditional non-restoring algorithm for 32-bit operands that consumes a total of 203 FFs. When configured for identical operand widths, the algorithm presented in this work consumes a total of 169 FFs which is an insignificant difference when considering the abundance of this

```

1  s_busy <= '1' when (s_req='1' or s_old_ack='1' or s_last_o_done='0')
2      else '0';
3
4  req_proc : process (i_clk_1)
5  begin
6      if rising_edge(i_clk_1) then
7          if (s_busy='0' and i_load_1='1') then
8              s_req <= '1';
9          elsif (s_old_ack='1') then
10             s_req <= '0';
11         else
12             s_req <= s_req;
13         end if;
14     end if;
15 end process req_proc;
16
17 reqpipe_proc : process (i_clk_2)
18 begin
19     if rising_edge(i_clk_2) then
20         s_xreq_pipe <= s_xreq_pipe(1 downto 0) & s_req;
21         s_new_req <= s_xreq_pipe(2);
22         s_last_req <= s_new_req;
23     end if;
24 end process reqpipe_proc;
25
26 ackpipe_proc : process (i_clk_1)
27 begin
28     if rising_edge(i_clk_1) then
29         s_xack_pipe <= s_xack_pipe(1 downto 0) & s_last_req;
30         s_old_ack <= s_xack_pipe(2);
31     end if;
32 end process ackpipe_proc;
33
34 loaddata_1_proc : process (i_clk_2)
35 begin
36     if rising_edge(i_clk_2) then
37         if (s_last_req='0' and s_new_req='1') then
38             s_data0_1 <= i_data0_1;
39             s_data1_1 <= i_data1_1;
40             s_data2_1 <= i_data2_1;
41             s_data3_1 <= i_data3_1;
42             s_start <= '1';
43         else
44             s_data0_1 <= s_data0_1;
45             s_data1_1 <= s_data1_1;
46             s_data2_1 <= s_data2_1;
47             s_data3_1 <= s_data3_1;
48             s_start <= '0';
49         end if;
50     end if;
51 end process loaddata_1_proc;

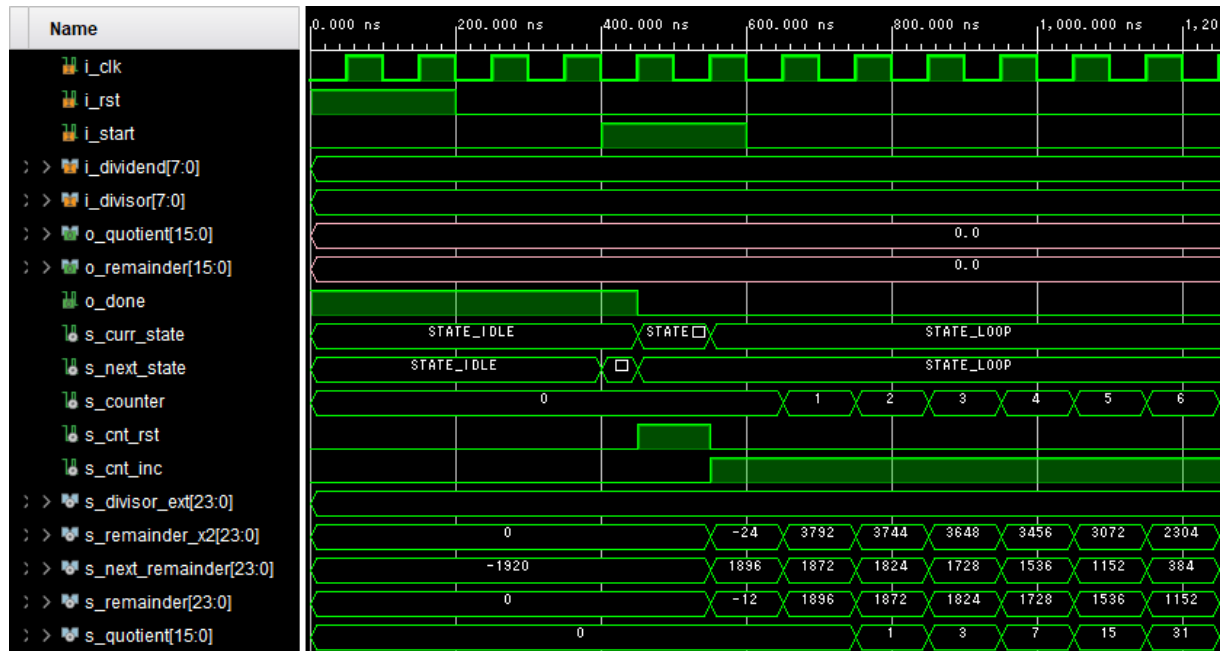
```

Listing 9: LASER2SVM Module's CDC Handshake implementation.

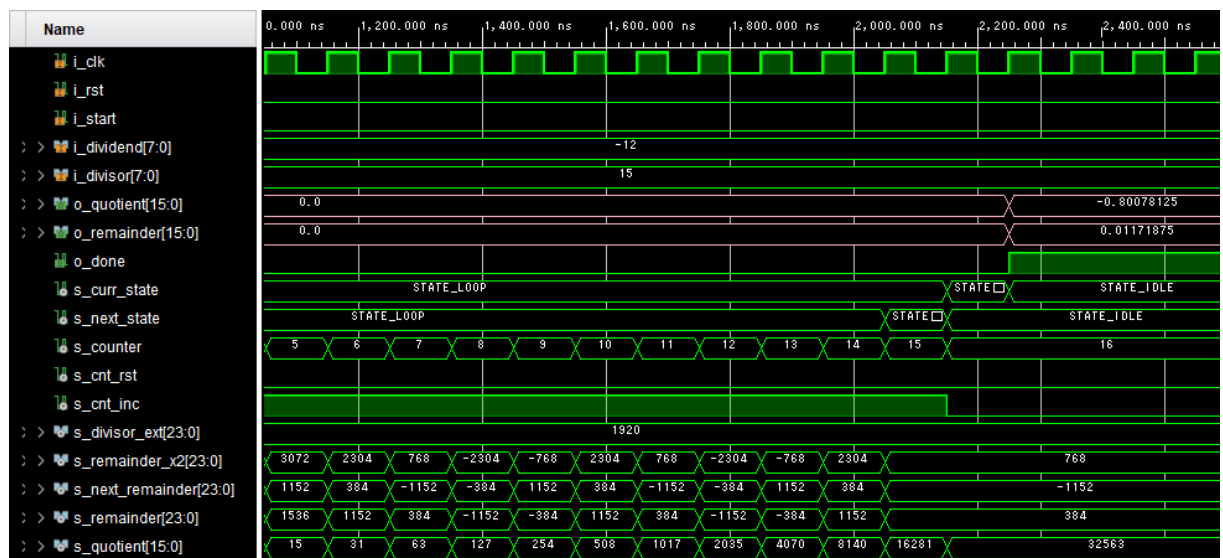
resource in FPGA devices. Their algorithm requires a total of 32 clock cycles to produce the result, while the divider proposed in this work requires 34 instead, once again confirming that these are comparable implementations, with minor differences. These same authors have produced extensive research on the design of arithmetic circuits for FPGAs, having proposed different approaches to implement non-restoring division circuits. These include decimal dividers that eliminate the need for binary coding/decoding circuitry, leading to major improvements in latency for larger operands [60], as well as both pipelined (favouring higher throughput) and combinational approaches (minimizing total latency) [61]. In this work, however, the most traditional approach provides reduced resource consumption while still providing acceptable latency results given the much longer realtime deadline in consideration. In Section 5.3.2, an RCS architecture is proposed in which the LASER2SVM module is successfully instantiated at a clock frequency of 50 MHz. Using the operand and output width configuration presented in Listing 8, the total latency (when ignoring the contributions of the CDC mechanism) is 540 ns, which compares favourably to the total budget of 10ms for new classification results to be produced and transmitted via CAN by the system. The testing results of this module, as seen from Vivado's Simulation window, are shown below in Figure 4.2 (split into two smaller Figures, 4.2a and 4.2b, for convenience). The module was configured for an input width of 8-bits and an output width of 16-bits, meaning that the output will be produced with 8 integer and 8 fractional bits. The registers containing the intermediate values produced in the "STATE_LOOP" stage show that the divisor input value has been multiplied (or, more accurately, left-shifted) to ensure the correctness of the algorithm, as previously explained. The final quotient result is -0.80078125 with a remainder of 0.01171875, which evidences that this number format has finite accuracy which produces round-off errors that are propagated and aggravated with every new arithmetic operation. The simulated waveforms showcase the linear relationship between quotient bits and computational time-complexity.

4.4 Discussion

The proposed modules achieve adequate simulation results and have moderate resource usage, but it is not possible to determine their adequacy for the RCS system. The main issue is that the CDC mechanism requires more testing and validation than what could be done for this work, to formally validate its functionality. Xilinx's "Design Analysis and Closure Techniques" user guide [62] provides some documentation on how to use their analysis tools to verify that CDC structures are all safe and properly constrained. However, the produced division module is quite simple and easily configurable which should allow for quick iterations of the design using the same division module.



(a)



(b)

Figure 4.2: Divider Module's simulation results of division of -12 by 15, with P=8.

Chapter 5

Support Vector Machine

The centrepiece of the RCS is the machine learning algorithm that delivers road condition classifications. For this prototype as well as the previous one, the selected data model is a linear Support Vector Machine, a versatile classification algorithm. The execution of this model on the previous prototype exhausted all memory resources, and even smaller subsets of the algorithm could not meet the defined deadline (producing a new classification every 10 ms). It was safe to assume that it would take on a similarly critical role when gauging the success or failure of the new RCS prototype proposed in this document.

This Chapter details the system's machine learning algorithm, and is structured as follows: Section 5.1 details the original C code that was adapted for proposed design, and provides a brief overview on SVMs; Section 5.2 presents the proposed design, describing the complete datapath and associated control logic; Section 5.3 describes the implemented modules, detailing numerous design aspects like resource utilization, performance and accuracy; to finalize, Section 5.4 discusses the contents of the whole Chapter, highlighting the future work required to further validate and iterate upon the proposed system.

5.1 Analysis

In machine learning nomenclature, SVMs are defined as supervised learning models that are capable of both classification and regression data analysis. They're classified as supervised learning models as they require a dataset where data is labelled, and using the model usually involves two main phases: training and classification. Conception occurs in the training phase, where the model is built for correctly classifying future data samples based on the labelled training dataset provided. This capability is harnessed in the subsequent classification phase. The SVM model proposed in this system has already undergone the training phase, so its only necessary to implement the corresponding classification algorithm to be run on the new FPGA platform, with some minor adaptations.

The working principle of SVMs was first described in 1963, and is based on the concept of a decision

boundary that separates two distinct data classes in order to partition the training dataset with high accuracy [63]. The training phase produces the maximum-margin hyperplane (the hyperplane that maximizes the distance between data samples of different classes) that constitutes the decision boundary. This hyperplane is defined by a series of points called support vectors, circled below in Figure 5.1, adapted from [29].

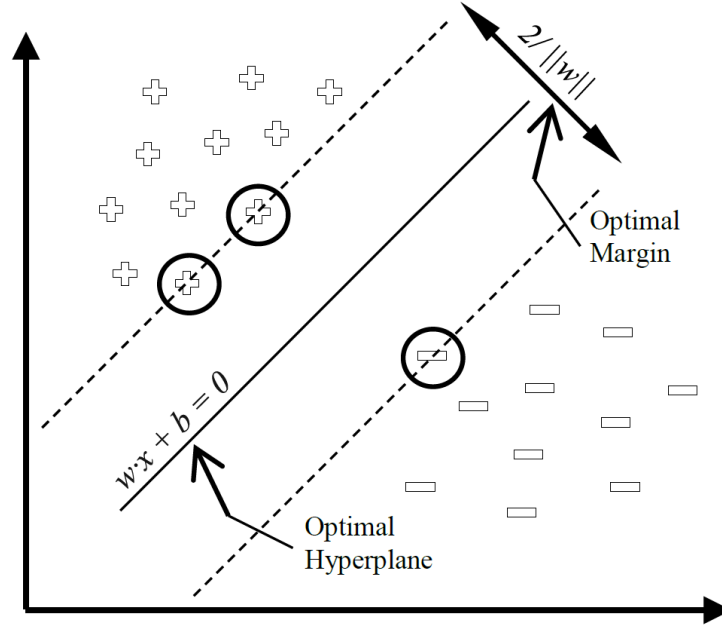


Figure 5.1: SVM hyperplane.

By the end of the training process, the portion of the data points not belonging to the decision boundary can be discarded, and only the support vectors are retained in the trained model. The original maximum-margin hyperplane algorithm constructed a linear classifier, like the one shown in Figure 5.1. In many real-world classification problems, it is not possible to linearly separate the training data in the original space. A new approach proposed in 1992 involved mapping the original input space into a higher dimensional one (by employing the “kernel trick”, proposed by Aizerman et al. [64]), in which a linear separation of data samples was possible [65]. The kernel employed for a given SVM model heavily dictates its computational requirements, as well as its potential accuracy in some applications. One of the simplest kernels available is the linear kernel, shown below in Equation 5.1.

$$linear_kernel_i = \sum_{j=0}^{N_{features}} support_vector_{i,j} \times feature_j \quad (5.1)$$

As Equation 5.1 shows, the number of kernels that must be computed equates the number “i” of support vectors defined during the classification stage. Each kernel is produced by a multiply-accumulate

operation of support vectors and a set of “j” features which constitute the input data to be classified. Other common SVM kernel are shown below:

$$poly_kernel_i = ((\gamma \times linear_kernel_i) + r)^d, \gamma > 0 \quad (5.2)$$

$$sigmoid_kernel_i = \tanh((\gamma \times linear_kernel_i) + r) \quad (5.3)$$

$$rbf_kernel_i = \exp(-\gamma \times (\sum_{j=0}^{N_{features}} (support_vector_{i,j} - feature_j)^2)) \quad (5.4)$$

The polynomial and sigmoid kernels (Equations 5.2 and 5.3, respectively) inherit part of their arithmetic operations from the linear kernel. On the other hand, the Gaussian Radial Basis Function (RBF) kernel (Equation 5.4) does not share any features with the other kernels. The kernel parameters γ , r and d must be defined by the model designer during the training procedure. Finally, a new data sample s is classified according to the output (sign) of the decision function (Equation 5.5):

$$F(x) = sign(\sum_{i=0}^{N_{SV}} \bar{a}_i y_i k(x_i, s) + \bar{b}) \quad (5.5)$$

In this Equation, $\bar{b}$ is an optional bias term, and the value of the $k(x_i, s)$ depends on the chosen kernel. The proposed system employs the linear kernel as it is the simplest of all the common kernels presented, and the proposed system aims to be a proof-of-concept, and not necessarily achieve the highest possible classification accuracies. Indeed, the proposed design could be considered a success if the accuracy achieved on the FPGA is close to the one achieved by the original classification algorithm, when taking into account any modifications made to adapt it for the new platform.

SVMs were originally described for binary classifications (where the training dataset's entries belong to one of two classes), and their use for multi-class classification is more complex. Several strategies have been proposed in literature, one of the most common being the “one-vs-one” approach [66] [67] that builds $k(k-1)/2$ classifiers where each one is trained on data from two classes. This is the method used by the algorithm described in this Section. After all classifiers are constructed, a voting strategy (proposed in [67]) is used to determine the final classification of a given input data, by selecting the class with the highest number of votes. In the case that two classes have the same number of votes then the one with the smaller index is selected — this approach may not always be a good strategy, but it is widely applied.

Each of the classifiers can be defined as a decision rule, which considers the “clash” between two

different classes, class “i” and “k” as shown in Equations 5.6, 5.7 and 5.8 below.

$$decision_rule_i = \sum_{j=k_start}^{k_end} kernel_j \times coef_{i,j} \quad (5.6)$$

$$decision_rule_k = \sum_{j=i_start}^{i_end} kernel_j \times coef_{k-1,j} \quad (5.7)$$

$$decision_rule_d = decision_rule_i + decision_rule_k \quad (5.8)$$

From within the total number of kernels produced, certain subsets are defined (as seen in the lower and upper bounds of summation in each Equation) which pertain to the decision boundaries between different class pairs. The value of $decision_rule_d$ is then compared to another value, defined as the intercept, to determine which class is the victor and will receive a “vote”, as shown below in Equation 5.9.

$$decision_rule_d < intercept_d \quad ? \quad \begin{cases} \text{if true, then class } i \text{ receives a vote.} \\ \text{if false, class } k \text{ receives a vote.} \end{cases} \quad (5.9)$$

This multi-class classification approach depends on additional structures, like the two-dimensional “coef” matrix with “N_CLASSES-1” columns and “N_VECTORS” rows, and the “intercept” array with $k(k-1)/2$ entries (with k being the number of classes of the model). The original C algorithm upon which this entire Chapter is based will be presented, decomposed in several parts and analysed, starting with the constant definitions shown in Listing 10 which provide an overview of the dimensions of the SVM model.

```

1  #define ^^IN_FEATURES      6
2  #define ^^IN_VECTORS      2533
3  #define ^^IN_CLASSES      4
4  #define ^^IN_INTERCEPTS  6

```

Listing 10: Constant definitions of SVM algorithm.

The “N_FEATURES” constant defines the number of values associated with each new data. In other words, it describes the number of data inputs of the proposed architecture: the six ratios associated with data samples of four distinct wavelength emissions (for convenience, lets adopt the labels w_0 , w_1 , w_2 and w_3 for each laser data sample at a given instance). The ratio between each pair of the data samples is

calculated (using the module described in Chapter 4) to produce six values, as defined by Equation 5.10.

$$r_0 = \frac{w_0}{w_1}; \quad r_1 = \frac{w_0}{w_2}; \quad r_2 = \frac{w_0}{w_3}; \quad r_3 = \frac{w_1}{w_2}; \quad r_4 = \frac{w_1}{w_3}; \quad r_5 = \frac{w_2}{w_3} \quad (5.10)$$

These six ratios constitute a single data input to be classified by the model. The “N_VECTORS” definition relates to the results obtained during the training phase, and more specifically the number of support vectors that define the classification boundary. “N_CLASSES” specified the number of different labels present in the training dataset, and consequently the number of valid classification results the algorithm produces.

The first loop of the algorithm is analogous to the operations described in Equation 5.1. As seen before, the number of kernels that must be calculated matches the number of support vectors defined during the training phase — for the RCS algorithm, a total of 2533 kernels must be calculated for each new classification.

```

1  for (i = 0; i < N_VECTORS; i++)
2  {
3      double temp = 0;
4      for (j = 0; j < N_FEATURES; j++)
5      {
6          temp += vector[i][j] * feature[j];
7      }
8      kernel[i] = temp;
9  }
```

Listing 11: Detail of Kernel Computation stage of the SVM algorithm.

Listing 12 handles the calculation of the decision values for each one-versus-one class confrontation, implementing the mathematical operations shown in Equations 5.6, 5.7 and 5.8.

Finally, the voting procedure (described in 5.9) is carried out for each class confrontation, and then the “victor” is selected as shown in Listing 13.

A single functional requirement was identified based on the analysis summarized in this Section. Namely, the SVM module must classify input data upon request, producing new classifications results after a certain number of clock cycles such that the system as a whole can deliver new classification results on the external CAN bus every 1 ms.

Perhaps more importantly, a set of non-functional requirements were identified that could maximize the changes of success of the data classification module. After all, the whole point of this system is to accomplish what the previous RCS prototype had failed to do, and clearly “the devil is in the details”. These non functional requirements proved instrumental during the conception of the SVM module, described in

```

1  unsigned char range[] = {0,210,581,1446};
2  ~~~
3  for (i = 0, d = 0, l = N_ROWS; i < l; i++) {
4      for (j = i + 1; j < l; j++) {
5          double temp = 0;
6
7          for (k = range[j]; k < range[j+1]; k++) {
8              temp += kernel[k] * coefficient[i][k];
9          }
10
11         for (k = range[i]; k < range[i+1]; k++) {
12             temp += kernel[k] * coefficient[j - 1][k];
13         }
14
15         if(temp + intercept[d] > 0) {
16             amount[i] += 1;
17         } else {
18             amount[j] += 1;
19         }
20         d = d + 1;
21     }
22 }

```

Listing 12: Detail of “one-vs-one” multi-class approach of the SVM algorithm.

```

1  int classVal = -1;
2  int classIdx = -1;
3  for (i = 0; i < N_CLASSES; i++) {
4      if (amount[i] > classVal) {
5          classVal = amount[i];
6          classIdx = i;
7      }
8  }
9  return classIdx;

```

Listing 13: Detail of voting procedure of the SVM algorithm.

the subsequent Sections 5.2 and 5.3:

1. Exploit the underlying FPGA device architecture in an efficient manner, by ensuring the critical performance and memory resources are adequately utilized according to the manufacturer’s specifications and recommendations;
2. Adapt the algorithm to allow for more efficient implementations in FPGA devices.

5.2 Design

This Section presents the solutions that enabled a full implementation of the algorithm using only the PL of the FPGA device. As shown in Listings 11 and 12 of the previous Section, the microcontroller-based prototype of the RCS relied on double-precision floating-point numbers for nearly all intermediate calculations. Floating-point arithmetic [68] represents numbers in scientific notation which gives them a much larger range but at the cost of some accuracy, when compared to what can be achieved with fixed-point notation using the same number of bits. Most modern general-purpose computers use floating-point representations, as these devices may be used for a wide spectrum of applications: for instance, a computer may at one point need to represent the Avogadro constant ($N_A = 6.02214076 \times 10^{23} \text{ mol}^{-1}$) and, at other times, the charge of an electron ($q_e = 1.60217662 \times 10^{-19} \text{ coulomb}$). This format is suitable for machine learning training on desktop or server machines with ample amounts of memory and processing power. However, it can prove very inadequate when considering resource-constrained embedded devices — indeed, the trained SVM model could not be run on the previous RCS prototype.

The FPGA-based architecture proposed here makes extensive use of the fixed-point format, marking the first adaptation of the original algorithm. Electing to implement a fixed-point datapath instead of floating-point one can help meet tight system requirements and performance specifications in a number of ways. Fixed-point implementations of a given algorithm usually present a substantial reduction in FPGA resource usage when compared to floating-point implementations, which inherently leads to lower static power consumption. Fixed-point designs may even achieve latency improvements, depending on the mapping of operations to the configurable logic resources. Certain applications that do not make use of the extended range achievable with floating-point arithmetic can achieve comparable results and accuracy when converted to fixed-point arithmetic.

When implementing a custom fixed-point datapath, it is important to be conscious of the number of integer and fractional bits in each function. Given the need to use the lowest amount of resources possible, it is not feasible to implement a datapath that supports the entire range of values possible. In fact, it would be impossible to produce a perfectly accurate system that supports divisions, as was explored in Section 4.1. Instead, it is preferable to provide mechanisms that deal with the possibility of overflow and perform iterations to ensure an acceptable trade-off between round-off errors, latency and resource usage is met. The proposed design implements a reduced precision adaptation of the algorithm, and was supported by a rudimentary study of the data structures required which will be explored in this Section. No formal study or data analysis was performed to determine if the datapath format selection was ideal, as

any reasonable selection that didn't result in a significant loss of information was sufficient for the intended proof-of-concept. This Chapter describes fixed-point properties using the Q format [69].

The first data structure studied was the support vector matrix. To better define the ideal number of integer and fractional bits used to represent the values in signed fixed-point format, the distribution of values was analysed (see Table 5.1). The support vector matrix has 2533 rows and 6 columns for a total of 15198 cells. The fourth column shows the number of support vectors that could not be represented accurately in a given format, based on the corresponding integer range. The analysis clearly shows the trade-off between the range of values representable and the resolution: for instance, the Q2.22 format achieves the highest resolution possible, but 19.13% of the support vectors would be saturated to fit the integer range indicated, implying a substantial loss of information. The implemented design employs Q6.18 as a common format to represent all support vectors, meaning the 3 outliers were saturated for the range of -32 to 31.

Format	Integer range	LSB value	Outliers	% Outliers
Q2.22	-2 to 1	2.38×10^{-7}	2907	19.13%
Q3.21	-4 to 3	4.76×10^{-7}	220	1.45%
Q4.20	-8 to 7	9.54×10^{-7}	15	0.10%
Q5.19	-16 to 15	1.91×10^{-6}	6	0.04%
Q6.18	-32 to 31	3.81×10^{-6}	3	0.02%

Table 5.1: Distribution of values of support vector matrix.

The training dataset used to create the SVM model for the RCS was not made available to the author for the development of this thesis. As such, the support vectors provided were the only known data entries that could be studied to inform the format selection for the data inputs of the algorithm. The data distribution is then identical to the previous one, with the caveat that the total number of bits is now 18 instead of 25. The format selected for the data inputs is Q8.9, which gives a range of -128 to 127 and a maximum resolution of 1.95×10^{-3} .

The Q format selection to represent the coefficients did not require the evaluation of several alternatives, as these values are extremely well suited for representation using fixed-point values. The coefficient matrix has 3 rows and 2533 columns, which add up to a total of 7599 elements. The magnitude of values of the entire set is contained in the smallest range possible (-2 to 1), which made selecting Q1.16 quite evident as it maximized the information retention with a resolution of 1.53×10^{-5} .

The functional block diagram of the proposed SVM design is presented below in Figure 5.2, and is

best analysed from left to right. A brief overview of the fixed-point datapath follows. The six data inputs of the system are to be implemented in Q8.9 format, which describes a 2's-complement fixed-point integer with 1 sign bit, 8 integer bits and 9 fractional bits for a total of 18-bits. These are fed to a systolic array structure that handles the kernel dot product operations. A set of six block RAM is also an input of the systolic array, outputting numbers representing the support vectors in the Q8.19 format. The output of the systolic array is in the Q20.27 format, and is directed to a saturation entity where it is transformed to Q8.16. It then serves a total of six multiply-accumulate units where the decision rules are computed, representing the "clash" between different pairs of classes. A set of BRAMs also connect to the MAC units, providing the needed coefficient values in the Q1.16 format. These values are fed to digital comparators that determine if the input is greater than the corresponding intercept value or not. The combined output of these comparators constitutes the SVM Module's output, as they encode the final classification.

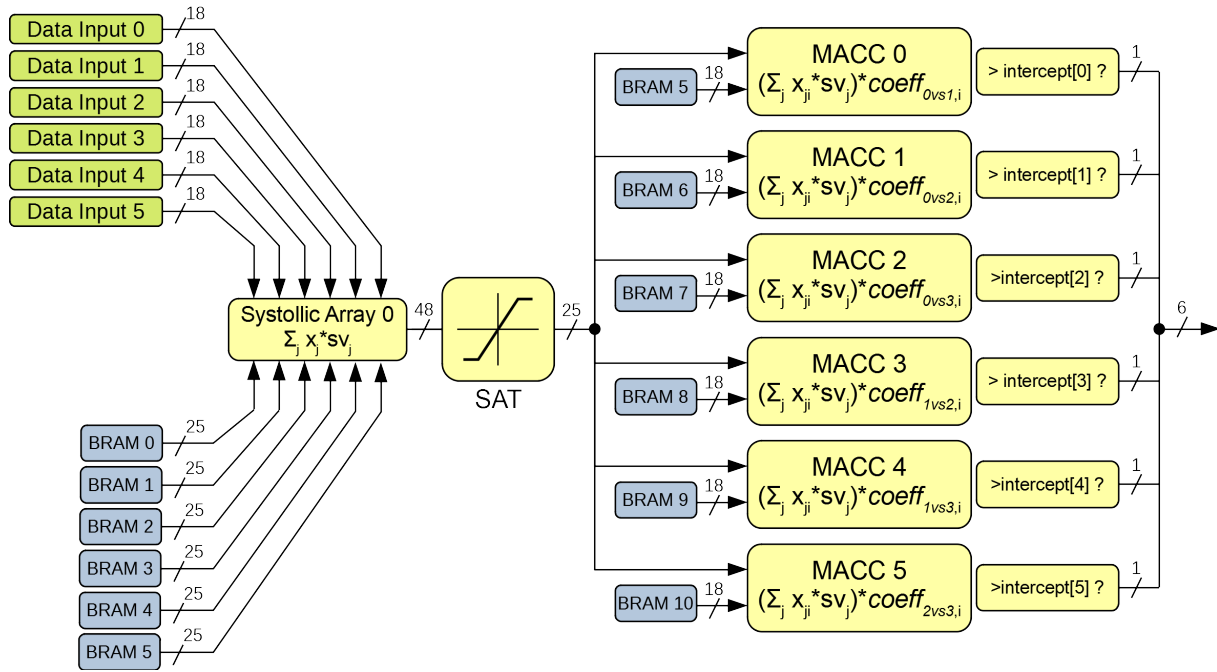


Figure 5.2: SVM Module's functional block diagram.

The systolic array is in charge of implementing Equation 5.1. To better understand the proposed design, an adaptation is presented in Figure 5.3 containing only two Processing Elements (PE). Each BRAM is loaded with support vectors values, where each BRAM 'i' ($0 \leq i < N_FEATURES$) is loaded with the values $sv[i][N]$ ($0 \leq N < N_VECTORS$). Given this specification, the final systolic array implementation will include a total of six DPS48 and BRAM blocks. However, as is explored in the next Section, a single BRAM is not capable of holding 2533 entries with the specified bit width and some adaptations must be made to the functional diagram. Each processing element has a latency of 3 clock cycles, giving the

systolic array a total of $6 \times 3 = 18$ clock cycles of latency, after which it achieves a throughput of 1 kernel computation per cycle. In other words, latency scales linearly with the number of input features, and the total computation time scales linearly with the number of support vectors. If these prove to be a limiting factor in performance, a different and more complex architecture could be defined where a set of systolic arrays operate in parallel — but, for the purposes of this work, the simplest route was taken. The control logic of the SVM module, composed of a set of counters and clock enable signals, ensure each submodule is activated when and for as long as required to perform the specified operations.

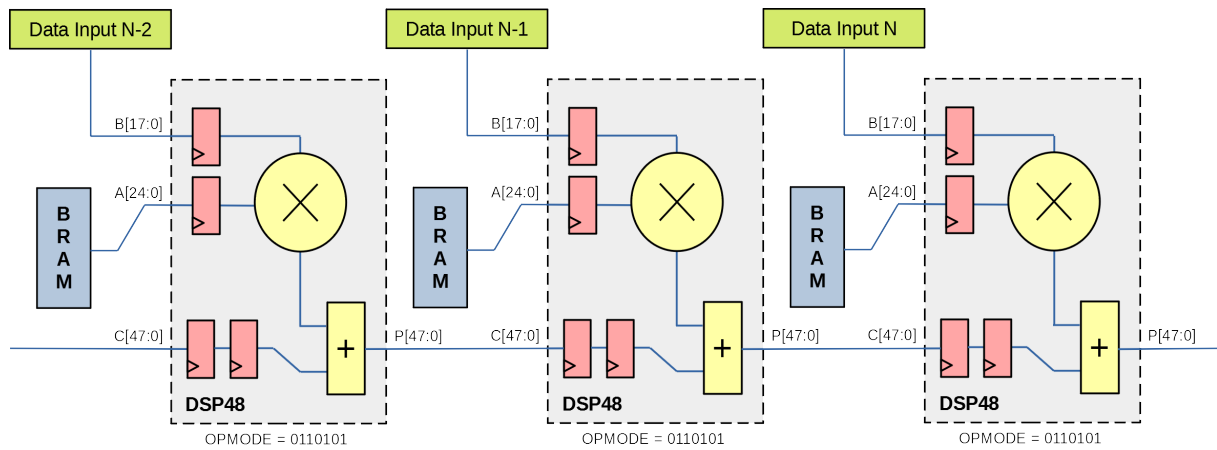


Figure 5.3: Detail of systolic array submodule.

The use of 18- and 25-bit widths for this custom datapath is due to the dimensions of the DSP48E1 slices, previously detailed in Section 2.2.3. Not exceeding the input widths specified will allow for lower resource usage and improved resource utilization. When higher operand widths are requested, the synthesis tools will infer a combination of multiple DSP48E1 slice to implement the desired function, requiring more of the scarce processing units available on the development board and almost certainly increasing latency. As was specified, the data inputs are 18-bits wide and the support vector values are 25-bits wide, matching the inputs of the internal multiplier of the DSP48E1 slice. To achieve this, the 18 Kb BRAM's configurable width word was used to specify 512 data entries of 36-bits, of which only 25 are used to store information. This design decision fully utilizes the operand widths of the dedicated multipliers, but wastes 30.5% of each BRAM's total capacity.

The output of the systolic array is fed to a saturation circuit that imposes an integer range of -512 to 511, which corresponds to the Q8.16 format specified. At first glance, the 6 MAC units represented in the block diagram seem to operate in parallel, but that is not the case. These units implement the decision rule described in the Equation 5.8, which depend on the multiply-accumulation operations described in Equations 5.6 and 5.7. The lower and upper bounds of the summation are contained in the “range”

variable, in line 1 of Listing 12. This practical consequences of this configuration is summarized below in Table 5.2.

Confrontation	Equation 5.6	Equation 5.6	Total
Class 0 vs. Class 1	coef[0][0] to coef[0][209]	coef[0][210] to coef[0][580]	581
Class 0 vs. Class 2	coef[1][0] to coef[1][209]	coef[0][581] to coef[0][1445]	1075
Class 0 vs. Class 3	coef[2][0] to coef[2][209]	coef[0][1446] to coef[0][2532]	1297
Class 1 vs. Class 2	coef[1][210] to coef[1][580]	coef[1][581] to coef[1][1445]	1236
Class 1 vs. Class 3	coef[2][210] to coef[2][580]	coef[1][1446] to coef[1][2532]	1458
Class 2 vs. Class 3	coef[2][581] to coef[2][1445]	coef[2][1446] to coef[2][2532]	1952

Table 5.2: Analysis of kernel coefficients required for each class confrontation.

As Table 5.2 shows, there is some imbalance in the computations required for each classification: for instance, assuming a completely sequential implementation, the “Class 2 vs. Class 3” computations would take nearly four times longer to complete than those of “Class 0 vs. Class 1”. It also has a direct impact on the number of coefficient values each BRAM must hold. The internal architecture of the MAC units is shown below in Figure 5.4.

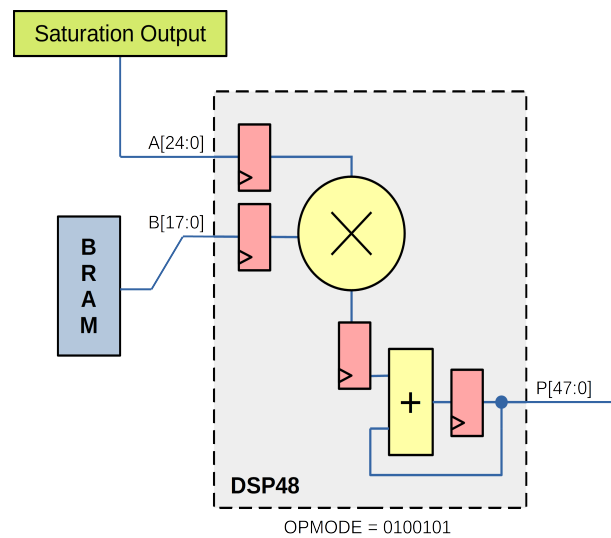


Figure 5.4: Detail of Multiplier-Accumulator (MAC) unit.

5.2.1 Test Cases

In this Section the test case for the SVM Module will be described. The goal of the tests described is to verify the arithmetic integrity of the module: not only should the classification results be mostly identical

to what is achieved using higher-precision algorithms run on a desktop machine, the magnitude of the round-off error should be analysed by comparing intermediate results between the SVM Module and the original algorithm.

However, it is important to note that the algorithm to be implemented is slightly altered, as output values of certain submodules must be saturated within a certain range of values when being introduced to following stages of the algorithm. To make accurate and fair comparisons, the saturation stages were added to the original algorithm (described in Section 5.1), as highlighted below in Listing 14. This change means that the implemented algorithm is fundamentally different than the original, and the classifications produced will not be accurate. However, demonstrating the equivalence between the new “saturated” algorithm run in a desktop environment using double precision numbers and the implemented custom-datapath FPGA algorithm would validate the approach.

```
1  for (i = 0; i < N_VECTORS; i++)
2  {
3      double temp = 0;
4      for (j = 0; j < N_FEATURES; j++)
5      {
6          if (vector[i][j] > 255) vector[i][j] = 255;
7          if (vector[i][j] < -256) vector[i][j] = -256;
8          temp += vector[i][j] * feature[j];
9      }
10     kernel[i] = temp;
11 }
```

Listing 14: Kernel value saturation, to ensure arithmetic equivalence between the original and the implemented SVM algorithm.

The SVM module will be integrated into a custom AXI memory-mapped IP using Vivado Design Suite's IP Packager, allowing for the necessary read/write operations on the SVM module's interface. A test application will be designed and run on either the ZYBO Z7-10's PS or a soft-processor instantiated in the design. This application will run classifications on the SVM Module, by loading its data inputs, initiating the classification procedures and retrieving the result after an adequate number of clock cycles. This application will run tests for all of the 2533 support vectors, which are the only data entries of the original dataset used to train the model that this work had access to. An identical C or C++ application will be run on a desktop machine using double-precision numbers and the results obtained will be compared, with the expectation being that these match the ones obtained when testing the SVM Module on the FPGA.

5.3 Implementation & Evaluation

Before discussing any architectural details of the implemented design, it is important to present the VHDL package defined that is used by several of the components of the SVM Module. It is presented below in Listing 15. A total of nine constants and five custom type definitions are included in the package. The first four constants (lines 2 to 5) are self evident, and match those detailed in the previous Section with the exception of “N_SUPPORT_VECTORS” which defines the number of support vectors used in each systolic array, and not the total amount of support vectors of the model. The “MA_LATENCY” is used for the implementation of the control logic, which employs a set of counters and clock enable signals that define when each computing element is active. The remaining four constants (lines 7 to 10) are used to define certain datapath bit widths, as one can verify by studying the custom types defined in the package (lines 12 to 21). The goal behind this package is to minimize design iteration and alteration efforts, as many of the submodules and data buses implemented rely on these auxiliary constructs.

```

1  package SVM_PACK is
2      constant N_SUPPORT_VECTORS : natural := 512;
3      constant N_FEATURES       : natural := 6;
4      constant N_CLASSES        : natural := 3;
5      constant N_INTERCEPTS   : natural := 6;
6      constant MA_LATENCY       : natural := 3;
7      constant DATA_WIDTH      : natural := 18;
8      constant SV_WIDTH         : natural := 25;
9      constant COEFF_WIDTH      : natural := 18;
10     constant INTERCEPT_WIDTH : natural := 25;
11
12     type DATA_ARRAY_type      is array (integer range <>)
13                                   of std_logic_vector(DATA_WIDTH-1 downto 0);
14     type SV_ARRAY_type        is array (integer range <>)
15                                   of std_logic_vector(SV_WIDTH-1 downto 0);
16     type PC_ARRAY_type        is array (integer range <>)
17                                   of std_logic_vector(47 downto 0);
18     type COEFF_ARRAY_type      is array (integer range <>)
19                                   of std_logic_vector(COEFF_WIDTH-1 downto 0);
20     type INTERCEPT_ARRAY_type is array (integer range <>)
21                                   of std_logic_vector(47 downto 0);
22 end SVM_PACK;
```

Listing 15: SVM Module’s VHDL package interface.

The SVM Module’s interface is quite compact, given the complexity implemented underneath. The “i_start” triggers the inner control circuit and marks the beginning of a new classification procedure. The “i_data” inputs are loaded to internal registers and, in time, the final result is made available on the “o_result” output bus. The “o_done” signal is active whenever the module is not computing a new data

classification, which means it can be used as an interrupt signal to an external hard- or soft-processor that would retrieve the result at the appropriate time.

```

1  entity svm_top is
2      port(
3          i_clk      : in  std_logic;
4          i_start    : in  std_logic;
5          i_data     : in  DATA_ARRAY_type(N_FEATURES-1 downto 0);
6          o_result   : out std_logic_vector(N_INTERCEPTS-1 downto 0);
7          o_done     : out std_logic
8      );
9  end svm_top;

```

Listing 16: SVM Module's top module VHDL interface.

While functionally identical, the implemented design underwent some changes between what was the defined in the previous Section (see Figure 5.2) and what is now presented in Figure 5.5.

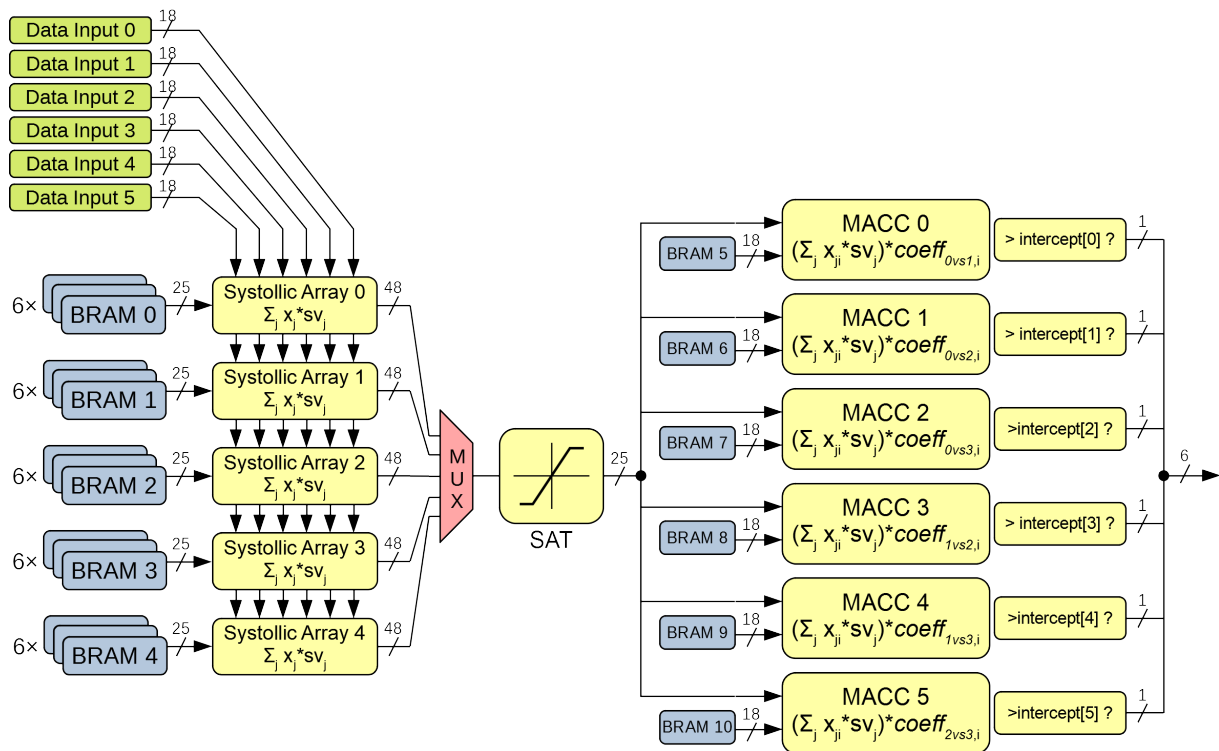


Figure 5.5: Block diagram of SVM Module's implementation.

The main difference from the initial design can be observed in the left side of Figure 5.5. There are now five systolic arrays each with an assigned BRAM, and the output of all systolic arrays is connected to a multiplexer before being fed to the saturation unit. These alterations imply an increased resource usage as well as power consumption. The goal behind them was to ensure optimal routing connections between memory and processing units to achieve the lowest possible latency. The main goal of this design is to

prove that the lengthy classification operations can be performed quickly, so this more “expensive” iteration was selected instead of the more “obvious” solution, in which a single systolic array and a collection of linked BRAM units would be used. Such a solution would require 30 less DSPs, and the author concedes it could be a much simpler, intuitive and resource-efficient design if achieving the lowest possible latencies (or indeed, the highest possible clock frequencies) was not strictly required. To better understand the way memory resources are distributed throughout the datapath, a summary of each BRAM’s content is provided below in Table 5.3 (which should be analysed in conjunction with Figure 5.5 that contains the numbered BRAM units). At the Table shows, most of the BRAMs hold 18 kB of memory, but some had to be configured for 36 kB given the asymmetry in the multi-class approach that was previously detailed in Table 5.2.

Name	Contents	Format	BRAM Configuration
BRAM 0	$sv[0]$ to $sv[580]$	Q6.18	6×18 kB; 512×36
BRAM 1	$sv[512]$ to $sv[1023]$	Q6.18	6×18 kB; 512×36
BRAM 2	$sv[1024]$ to $sv[1535]$	Q6.18	6×18 kB; 512×36
BRAM 3	$sv[1535]$ to $sv[2047]$	Q6.18	6×18 kB; 512×36
BRAM 4	$sv[2048]$ to $sv[2532]$	Q6.18	6×18 kB; 512×36
BRAM 5	$coef[0][0]$ to $coef[0][209]$, $coef[0][210]$ to $coef[0][580]$	Q1.16	18 kB; $1K \times 18$
BRAM 6	$coef[1][0]$ to $coef[1][209]$, $coef[0][581]$ to $coef[0][1445]$	Q1.16	36 kB; $2K \times 18$
BRAM 7	$coef[2][0]$ to $coef[2][209]$, $coef[0][1446]$ to $coef[0][2532]$	Q1.16	36 kB; $2K \times 18$
BRAM 8	$coef[1][210]$ to $coef[1][580]$, $coef[1][581]$ to $coef[1][1445]$	Q1.16	36 kB; $2K \times 18$
BRAM 9	$coef[2][210]$ to $coef[2][580]$, $coef[1][1446]$ to $coef[1][2532]$	Q1.16	36 kB; $2K \times 18$
BRAM 10	$coef[2][581]$ to $coef[2][1445]$, $coef[2][1446]$ to $coef[2][2532]$	Q1.16	36 kB; $2K \times 18$

Table 5.3: Analysis of BRAM contents in the final SVM Module’s implementation.

The post-implementation resource utilization is displayed below in Table 5.4. As would be expected, the SVM Module is the most resource demanding of all the modules described in this work, with a total

of 541 LUTs and 2205 FFs required. More importantly, there are 20 BRAM Tiles and 36 DSP48E1 Slices are in use. For larger SVM models that would exhaust all available BRAMs in the ZYBO Z7-10's PL, it would be necessary to employ the external DDR3 memories which would require a complete overhaul of the architecture proposed in this Section.

This resource usage compares favourably to a similar work developed by Ago et al. [70], who implemented an SVM model accepting 128 features and containing 95 support vectors. Their system also features a fixed-point datapath, albeit with a single format (Q3.14) used throughout. Their implementation consumes more hardware resources, requiring a total of 96 DSP48E1 slices and 100 (18 kB) BRAMs. However, their work achieved 331 clock cycles per computation, and these do not scale linearly with the number of support vectors of the model. Instead, the DSP slice and BRAM consumption seems to scale in a linear one-to-one manner with the support vectors. These comparisons further evidence the fact that implementing machine learning models on programmable hardware means optimizing for the desired trade-off between resource usage and performance.

Resource	Utilization	Available	Utilization %
LUT	541	17600	3.0738637
FF	2205	35200	6.2642045
BRAM	20	60	33.333336
DSP	36	80	45.0
IO	8	100	8.0
BUFG	1	32	3.125

Table 5.4: SVM Module resource utilization.

The design can be clocked at a maximum frequency of approximately 200 MHz, being capable of producing new classification results at a frequency of approximately 78 kHz. The classification time, measured from “i_start” activation until all compute units are done processing is 12785 ns, or 2557 clock cycles. This classification interval represents 0.12% of the defined deadline of 10 ms between classification results. In other words, results will be delivered at the desired cadence with plenty of time to spare. The precise timing characteristics depend on the way the Laser Controller Module is configured and at which clock frequency, but the available margin is decisively large enough.

5.3.1 Test Results

The system that supports the test applications on the ZYBO Z7-10 is shown below in Figure 5.6. It is an extremely simple design, where the SVM Module (integrated into an AXI wrapper module) is connected to the ZYNQ7 Processing System, where it can be easily controlled as a memory-mapped device.

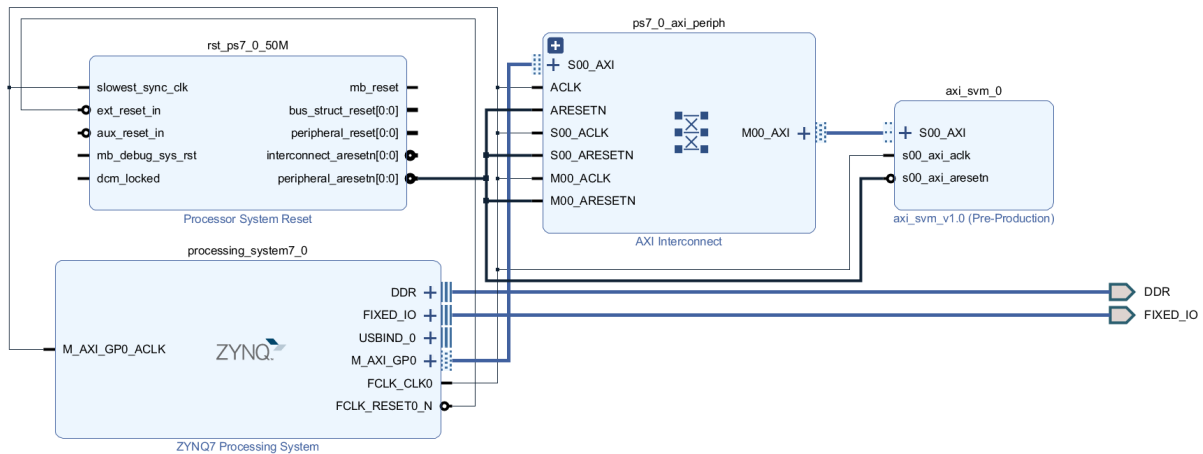


Figure 5.6: Block Design View of SVM Module AXI test.

The SVM Module was instantiated with a clock frequency of 50 MHz, as it shares its clock signal with the AXI wrapper's interface. This increases the computation time from the theoretical minimum of 12785 ns by a factor of four, meaning a new classification is now finished in 51140 ns. This is still a quite acceptable value, as it equates to only 0.48% of the defined deadline of 10 ms between classification results — which leaves 99.52% of the time budget free to the previous data acquisition and processing procedures (see Chapters 3 and 4), and the subsequent stage of communicating the classification result via the CAN bus lines. The main loop of the testing application is presented below in Listing 17.

The evaluation of test results compared the final value at the multiply-accumulator outputs after each classification, where the testing dataset was composed of the 2533 support vectors. The results show that 197 out of the 2533 classifications performed did not match the expected value, which translates to an error rate (not to be confused with the model's prediction accuracy, which is not discussed in this work) of 7.77%. These results suggest that this specific classification problem does not benefit greatly from double-precision floating-point numbers, and that a much more efficient and lightweight implementation using a custom fixed-point datapath can achieve similar levels of accuracy.

Some research work has been performed to assess the robustness of SVM algorithms when approximations caused by fixed-point and finite register widths are introduced. Evaluating the results obtained by the proposed architecture against these works makes for a fairer comparison, when compared to strictly

```

1  for (int i = 0; i<2533; i++)
2  {
3      Xil_Out32(svmPtr+4*1, sv[i][0]);          // i_data(0)
4      usleep(100);
5      Xil_Out32(svmPtr+4*2, sv[i][1]);          // i_data(1)
6      usleep(100);
7      Xil_Out32(svmPtr+4*3, sv[i][2]);          // i_data(2)
8      usleep(100);
9      Xil_Out32(svmPtr+4*4, sv[i][3]);          // i_data(3)
10     usleep(100);
11     Xil_Out32(svmPtr+4*5, sv[i][4]);          // i_data(4)
12     usleep(100);
13     Xil_Out32(svmPtr+4*6, sv[i][5]);          // i_data(5)
14     usleep(100);
15     // Start processing
16     Xil_Out32(svmPtr+4*0, 0x0001);            // i_start = '1';
17     usleep(100);
18     Xil_Out32(svmPtr+4*0, 0x0000);            // i_start = '0';
19     usleep(100);
20     // Read classification result
21     tempData = Xil_In32(svmPtr+4*7);
22     usleep(100);
23     res[i] = tempData;
24 }

```

Listing 17: Detail of the SVM Module's C++ testing application deployed on the ZYBO Z7-10's PS.

measuring the loss of classification accuracy which cannot be gauged here. Anguita et al. [71] have studied these effects on the learning phase of the SVM algorithm using datasets often used in academic research of machine learning. Using the IRIS dataset, they studied the error rate by measuring the Euclidean distance between the results obtained using equivalent fixed-point and floating-point solutions. The results show that using 12- to 18-bit register widths on the fixed-point architecture was enough to achieve an error rate of 0.08%, and that greater widths produce diminishing returns. Another set of tests was performed with the SONAR dataset, by comparing the number of misclassified patterns by their solution when using different register widths. The authors concluded that using 8- to 24-bit register widths along their datapath was sufficient to equal the accuracy of the floating-point SVM. In a later work, Anguita et al. [72] continued their research on the topic but focussed on an automotive application by employing the Daimler-Chrysler dataset, used to train machine learning algorithms in detecting the presence of pedestrians crossing a road. In summary, they conclude that representing coefficients, kernel and data values using 12-, 8- and 4-bit fixed-point values is sufficient to equal the performance of double-precision floating-point implementations. In the works cited previously, the authors have noted that the choice of fixed-point format is of crucial importance, and that it must be evaluated on a problem-to-problem basis. They often employ normalization techniques to ensure that a certain set of values can be adequately stored using fixed-point

formats with reduced integer bits. They also propose their own SVM algorithms which they demonstrate to be more resistant to quantization effects, and thus more suitable for FPGA implementations. Both of these techniques could allow for future improvements on the currently obtained 7.77% error rate of the proposed architecture.

5.3.2 RCS2FPGA Architecture

To further evaluate the feasibility of an FPGA-based implementation of the RCS device, a “mock-up” system was developed in Vivado Design Suite’s IP Integrator, to evaluate resource needs and detect possible implementation issues.

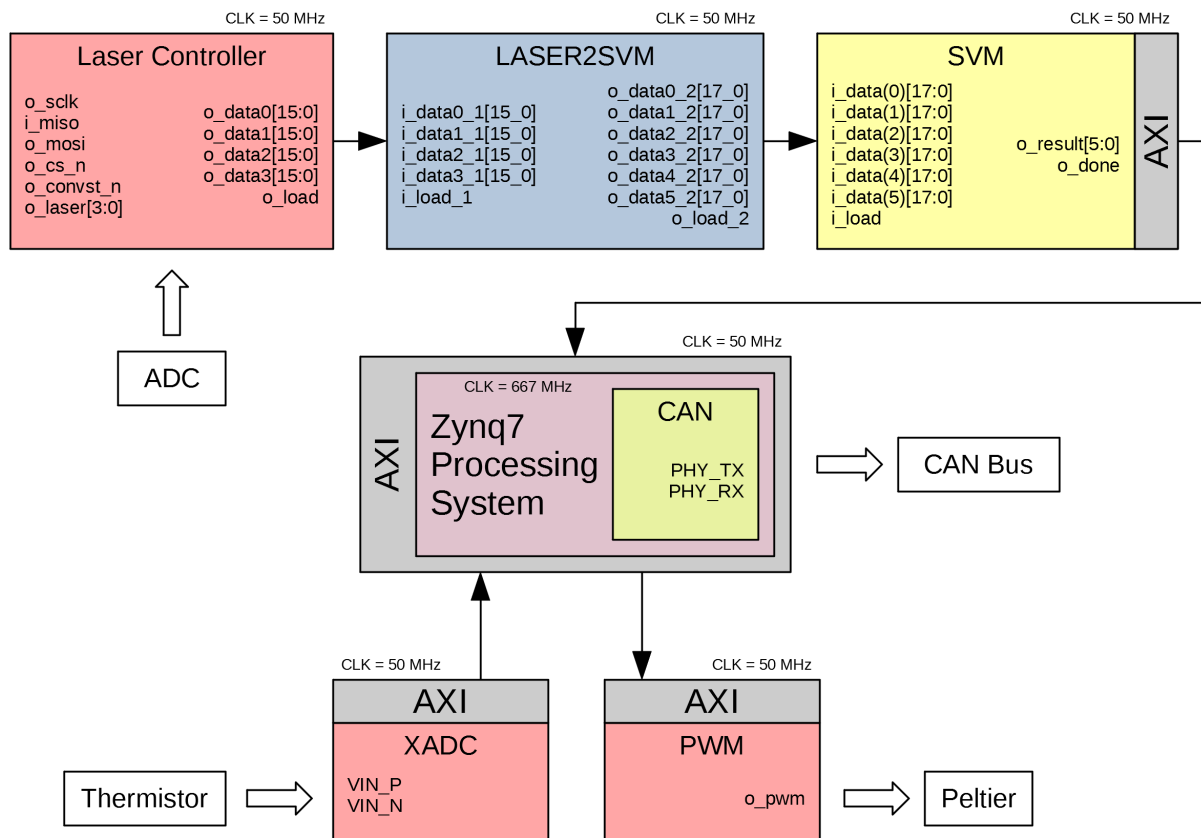


Figure 5.7: Functional Block Diagram of RCS2FPGA architecture.

The ZYBO Z7-10’s XADC was leveraged for the proposed thermal management solution, where it can be used to periodically sample voltage values derived from a simple voltage-divider assembly using the NTC thermistor, previously detailed in Section 2.2.5.2. A simple software-based PID controller was implemented on the board’s PS, containing random weight values and producing a control variable between 0 and 255. This value could then be written into an internal register of a simple AXI-compatible PWM generator, which

would produce a pulse-modulated signal intended for the Peltier Module (see Section 2.2.6) that would perform the necessary corrective action. This simple design serves a illustrative purpose, and any practical implementations would require careful calibration of the PID controller's weights and, evidently, testing with the actual system in operation. The ADT7301 is omitted in this prototype, but as the resource utilization shows there is ample room for a dedicated ADT7301 Module to be developed and deployed on the system (much in the same vein as the ADS8332 Module described in Section 3.2.2) to perform the desired temperature data readouts, with a minimal load on the main processor.

The implementation and deployment of a generic CAN Controller module onto a FPGA design is not as straightforward as the task of integrating an SPI, UART or I²C driver. This type of controller is most often used in the automotive industry, where its fault tolerance and error detection capabilities are of utmost importance when used for interfacing with critical systems components. These features also make it a much more complex protocol than those mentioned previously, greatly increasing the development time and testing efforts if the module was to be developed from scratch. Furthermore, even if when a self-developed CAN module is available, Bosch holds the patent to this technology and anyone wishing to implement and advertise ASIC- or FPGA-based systems as CAN-capable must pay a fee for the CAN Protocol License [73]. Despite Xilinx's ample library of generic IPs that can greatly reduce the burden on system developers, their CAN IP core is also not freely available for deployment. The only cost- and time-sensitive option to produce a CAN-capable system using the tools described in this project is to make use of the chosen development board's processing system which comes with a ready-to-use CAN controller peripheral (see Figure 2.1). The post-implementation resource utilization is displayed below in Table 5.5.

Resource	Utilization	Available	Utilization %
LUT	1833	17600	10.414772
FF	4361	35200	12.389205
BRAM	20	60	33.333336
DSP	36	80	45.0
IO	14	100	14.0
BUFG	1	32	3.125

Table 5.5: RCS2FPGA Architecture's resource utilization.

It is once more made clear that the machine learning algorithm is the "centrepiece" of the system, as it absorbs most of the hardware resources as well as the data processing time budget. It is responsible for

29.5% of the LUT utilization, 50.7% of the FF utilization as well as the 20 BRAMs and 36 DSPs present in the proposed architecture. This is further evidenced by Figures 5.8 and 5.9 which highlight the resources consumed by instantiating the SVM Module and the RCS2FPGA proposed architecture, respectively.

The results obtained suggest a final implementation of the system could forego the Zynq's PS as the main control unit, opting instead for a soft-processor deployed on the PL. This reasoning makes two basic assumptions on the hypothetical iteration of the system: firstly, that licensing and deployment of a CAN core implemented fully on the programmable logic of the device is taken care off; and secondly, that the burden on the main control unit (which is the software control application to be developed) can be supported by a soft-processor. This is because the maximum performance and memory resources which characterize a soft-processor that relies solely on the Zynq's PL could never exceed those possible using a dedicated, hard-processor such as the Zynq's PS. The author denotes that producing an entire system using only programmable logic resources can reduce costs and PCB footprint of the final system, which are highly desirable characteristics in an embedded system with packaging constraints such as the Road Condition Sensor (RCS). However, this comes with higher code development and maintenance costs, as an entire set of tools to develop, program and debug the soft CPU core and the software application must be procured.

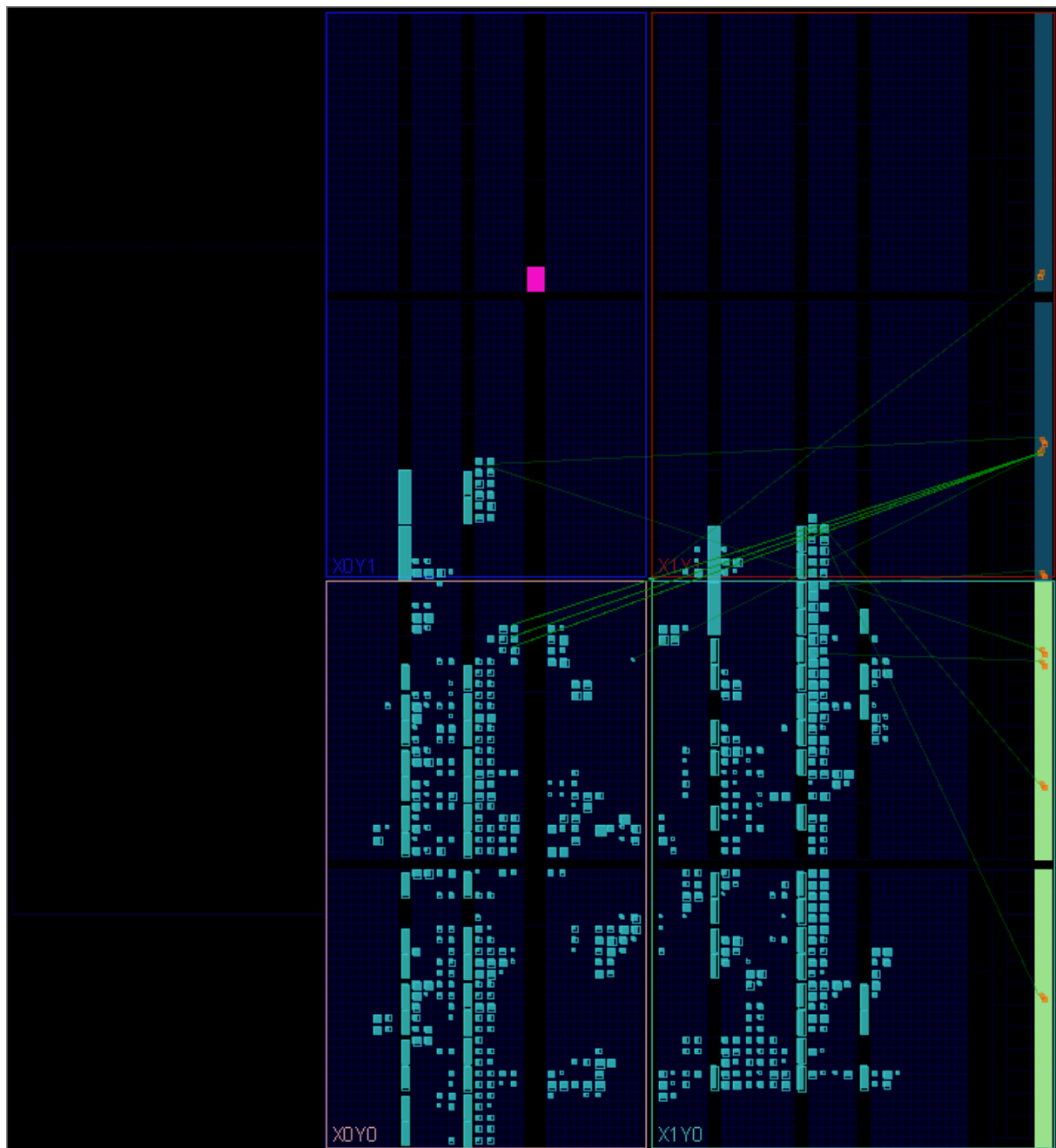


Figure 5.8: “Device View” of SVM Module’s implementation.

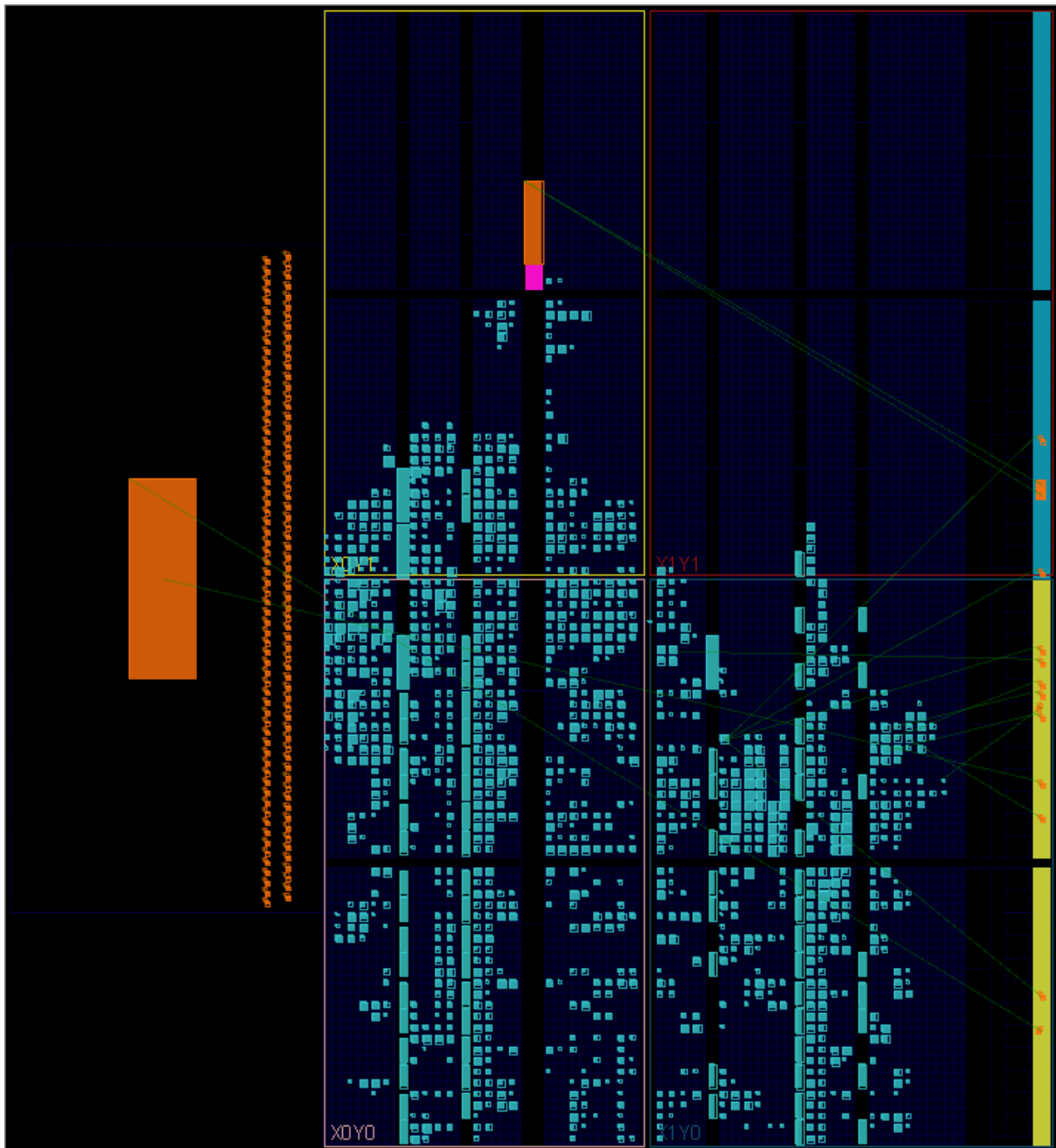


Figure 5.9: "Device View" of RCS2FPGA system implementation.

5.4 Discussion

The results obtained suggest that using an FPGA-based solution for the RCS project is a valid approach, as the proposed design shows that the most critical component of the previous prototype can be implemented very efficiently in hardware. The described architecture is capable of new data classifications in a small fraction of the available time budget, even when running at one fourth of its maximum operating clock frequency (consuming 0.48% of the target deadline with an input clock of 50 MHz). The low resource utilization further validates this approach, specially when considering that the system was implemented in the smallest device of the Zynq-7000 SoC family.

However, some problems were identified in the architecture and must be addressed by those wishing to implement such a system. Firstly, the SVM model would need to be re-trained to account for the additional saturation stages before being ported to the FPGA. Still, it is possible that this problem could have been averted during the data analysis stages which preceded the creation of the algorithm: Table 5.1 indicates the presence of outliers in the training dataset, which frequently originate from measurement errors and could have been excluded. Further information about the dataset would be necessary to verify this assessment. Another major issue of this work is that there is no proposed design for implementing other common SVM kernels, which can often achieve better results than the linear kernel in a number of data analysis problems.

Chapter 6

Conclusion and Future Work

Current approaches for implementing sophisticated sensor systems for the automotive industry are not always suitable to the new demands and challenges ushered in by the new generation of autonomous vehicles. In this sense, this thesis explored the implementation of a road condition sensor's control and data processing circuitry in an FPGA solution, in hopes of answering questions regarding the main challenges and opportunities associated with this approach. This Chapter reflects on the knowledge acquired during this work, and highlights some opportunities for expanding and indeed implementing such a system.

Firstly, the implemented Laser & Photodetector Controller is an example that a reconfigurable hardware platform provides the necessary flexibility and granularity to build high-performance and compact I/O communication protocols. Designers can produce systems that suit the exact I/O needs of the application, which can prove critical for applications that require numerous and/or I/O peripheral controller instances or those that can benefit from fine-grained realtime timing characteristics.

Secondly, the effort and care required to implement division on programmable hardware proved greatly superior than what is possible in the typical workflow for applications running on generic processors. The same can be said for having different modules within a system running at different clock frequencies: an issue that is seldom encountered when developing applications for microcontrollers and processors becomes a real headache for FPGA system designers.

Finally, the potential of this platform was more profoundly explored when developing the SVM architecture. While the design is presented as a mere proof-of-concept, the results obtained show that custom fixed-point datapath for SVM inference may equal or surpass a double-precision algorithm running on a microcontroller in both performance and memory footprint in suitable applications.

6.1 Future Work

Despite delivering on its promise to gauge the applicability of FPGA platforms for automotive sensor systems, this work has encountered some obstacles and limitations that must be understood by those wishing to implement similar devices.

Several of the problems relate to the complex process of developing and validating FPGA-based systems and hardware modules in general. For industrial applications, time-to-market is a key metric that could be compromised when moving to configurable hardware platforms. The proposed designs were not developed with security or reliability in mind, but evidently these aspects must be taken into account and will drive up the cost, system complexity and consequently the development time of the system. These considerations extend to the implementation of machine learning algorithms with custom fixed-point datapaths, as these number formats have inherently limited range and accuracy and operations introduce round-off errors and loss of significance. These effects should be formally evaluated using numerical analysis tools, and the algorithm may require some adaptations before being implemented. The CDC mechanisms employed by the proposed system also require specific validation techniques, for the problem of metastability is unpredictable and may not even be encountered using conventional testing methods.

However, other issues pertaining to the specific design choices made in this work were identified. For instance, the proposed SVM classifier design is not conducive to the inherent iteration process in engineering. That is, changes in the classification model may require substantial adaptations of the architecture — a problem that could be countered by adopting high-level synthesis tools and other utilities often provided by FPGA device vendors. The ambitious fully-synchronous design of the Laser Controller is also notably less flexible, and a more traditional approach relying on asynchronous event signalling or the ADC's auto-sampling features may facilitate future work.

Bibliography

- [1] Jennifer Shuttleworth. 'SAE Standards News: J3016 automated-driving graphic update'. In: *SAE International* (7th Jan. 2019). URL: <https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic> (accessed on 27/08/2020).
- [2] J. Simões et al. 'Evolution of the cruise control'. In: *International Conference on Regional Triple Helix Dynamics (HELIX2016)*. 2016.
- [3] C. Keatmanee et al. 'Vision-Based Lane Keeping - A Survey'. In: *2018 International Conference on Embedded Systems and Intelligent Technology International Conference on Information and Communication Technology for Embedded Systems (ICESIT-ICICTES)*. 2018, pp. 1–6. DOI: 10.1109/ICESIT-ICICTES.2018.8442051.
- [4] Pete Bigelow. 'Why Level 3 automated technology has failed to take hold'. In: *Automotive News* (21st July 2019). URL: <https://www.autonews.com/shift/why-level-3-automated-technology-has-failed-take-hold> (accessed on 15/09/2020).
- [5] Chris Ziegler. 'GM aims to speed up self-driving car development by buying Cruise Automation'. In: *The Verge* (11th Mar. 2016). URL: <https://www.theverge.com/2016/3/11/11195808/gm-cruise-automation-self-driving-acquisition> (accessed on 27/08/2020).
- [6] Brian Salesky. 'Why We Created Argo AI'. In: *Argo AI* (10th Feb. 2017). URL: <https://www.argo.ai/2017/02/why-we-created-argo-ai/> (accessed on 27/08/2020).
- [7] Brian Salesky. 'Our New Partner: Volkswagen AG Invests \$2.6 Billion at More Than \$7 Billion Valuation; Facilitates Expansion to Europe'. In: *Argo AI* (12th July 2019). URL: <https://www.argo.ai/2019/07/our-new-partner-volkswagen-ag-invests-2-6-billion-at-more-than-7-billion-valuation-facilitates-expansion-to-europe/> (accessed on 27/08/2020).
- [8] 'Waymo's fully self-driving vehicles are here'. In: *Waymo* (7th Nov. 2017). URL: <https://blog.waymo.com/2019/08/waymos-fully-self-driving-vehicles-are.html> (accessed on 27/08/2020).

- [9] 'Waymo reaches 5 million self-driven miles'. In: *Waymo* (27th Feb. 2018). URL: <https://blog.waymo.com/2019/08/waymo-reaches-5-million-self-driven.html> (accessed on 27/08/2020).
- [10] Rita Liao. 'Search giant Baidu has driven the most autonomous miles in Beijing'. In: *Tech Crunch* (2nd Apr. 2019). URL: <https://techcrunch.com/2019/04/02/baidu-self-driving-2018/> (accessed on 27/08/2020).
- [11] Shunsuke Tabeta. 'Baidu builds world's largest self-driving R&D center'. In: *Nikkei Asian Review* (28th May 2020). URL: <https://asia.nikkei.com/Business/China-tech/Baidu-builds-world-s-largest-self-driving-R-D-center> (accessed on 27/08/2020).
- [12] Danny Shapiro. 'How NVIDIA DRIVE PX Will Help Automakers Slim Down Self-Driving Cars'. In: *Nvidia* (17th Mar. 2015). URL: <https://blogs.nvidia.com/blog/2015/03/17/nvidia-drive-px/> (accessed on 27/08/2020).
- [13] Bob Sherbin. 'Live: NVIDIA's Las Vegas CES Press Event'. In: *Nvidia* (3rd Jan. 2015). URL: <https://blogs.nvidia.com/blog/2016/01/03/ces-las-vegas-event/> (accessed on 27/08/2020).
- [14] Danny Shapiro. 'NVIDIA and Bosch Announce AI Self-Driving Car Computer'. In: *Nvidia* (16th Mar. 2017). URL: <https://blogs.nvidia.com/blog/2017/03/16/bosch/> (accessed on 27/08/2020).
- [15] Danny Shapiro. 'Audi's New A8 Turns Mobility Into Magic, Using NVIDIA Tech to Transform Transportation'. In: *Nvidia* (11th July 2017). URL: <https://blogs.nvidia.com/blog/2017/07/11/audi-2018-a8-nvidia-barcelona/> (accessed on 27/08/2020).
- [16] 'The Evolution of EyeQ'. In: *Mobileye* (). URL: <https://www.mobileye.com/our-technology/evolution-eyeq-chip/> (accessed on 27/08/2020).
- [17] 'Industry First'. In: *Mobileye* (). URL: <https://www.mobileye.com/about/industry-firsts/> (accessed on 27/08/2020).
- [18] Paul Lienert Tova Cohen Ari Rabinovitch. 'Intel's \$15 billion purchase of Mobileye shakes up driverless car sector'. In: *Reuters* (13th Mar. 2017). URL: <https://de.reuters.com/article/uk-intel-mobileye/intels-15-billion-purchase-of-mobileye-shakes-up-driverless-car-sector-idUKKBN16K10V> (accessed on 27/08/2020).

- [19] Heather Somerville. 'Uber's self-driving unit valued at \$7.25 billion in new investment'. In: *Reuters* (19th Mar. 2019). URL: <https://www.reuters.com/article/us-uber-softbank-group-selfdriving/ubers-self-driving-unit-valued-at-7-25-billion-in-new-investment-idUSKCN1RV01P> (accessed on 27/08/2020).
- [20] Steve Trousdale. 'GM invests \$500 million in Lyft, sets out self-driving car partnership'. In: *Reuters* (5th Jan. 2016). URL: <https://www.reuters.com/article/us-gm-lyft-investment/gm-invests-500-million-in-lyft-sets-out-self-driving-car-partnership-idUSKBN0UI1A820160105> (accessed on 27/08/2020).
- [21] Joseph White. 'Ford, Lyft will partner to deploy self-driving cars'. In: *Reuters* (27th Sept. 2017). URL: <https://www.reuters.com/article/us-ford-motor-lyft/ford-lyft-will-partner-to-deploy-self-driving-cars-idUSKCN1C209L> (accessed on 27/08/2020).
- [22] Heather Somerville. 'Lyft surpasses 5,000 self-driving rides with Aptiv fleet'. In: *Reuters* (21st Aug. 2018). URL: <https://www.reuters.com/article/us-lyft-selfdriving/lyft-surpasses-5000-self-driving-rides-with-aptiv-fleet-idUSKCN1L61AX> (accessed on 27/08/2020).
- [23] Bjorn Carey. 'Stanford, Toyota to collaborate on AI research effort'. In: *Stanford News* (4th Sept. 2015). URL: <https://news.stanford.edu/2015/09/04/toyota-stanford-center-090415/> (accessed on 27/08/2020).
- [24] Adam Conner-Simons. 'CSAIL joins with Toyota on \$25 million research center for autonomous cars'. In: *MIT News Office* (4th Sept. 2015). URL: <https://news.mit.edu/2015/csail-toyota-25-million-research-center-autonomous-cars-0904> (accessed on 27/08/2020).
- [25] Adam Conner-Simons. 'Pioneering project releases Final Report'. In: *UK Autodrive* (24th May 2019). URL: <http://www.ukautodrive.com/final-report/> (accessed on 27/08/2020).
- [26] Dean Pomerleau. 'ALVINN: An Autonomous Land Vehicle In a Neural Network'. In: *Proceedings of Advances in Neural Information Processing Systems 1*. Ed. by D.S. Touretzky. Morgan Kaufmann, Dec. 1989.
- [27] Deva Ramanan. 'Pushing the Self-Driving Frontier: Argo AI Partners with Carnegie Mellon to Form Autonomous Vehicle Research Center'. In: *Argo AI* (24th June 2019). URL: <https://www.argo.ai/2019/06/pushing-the-self-driving-frontier-argo-ai-partners->

- with-carnegie-mellon-to-form-autonomous-vehicle-research-center/
(accessed on 27/08/2020).
- [28] José Varela Rodrigues. 'Carro autónomo: Governo e Bosch fecham acordo de 35 milhões para desenvolver soluções em Braga [Autonomous car: Government and Bosch close 35 million deal to develop solutions in Braga]'. In: *Jornal Económico* (2nd Apr. 2019). URL: <https://jornaleconomico.sapo.pt/noticias/carro-autonomo-governo-e-bosch-fecham-acordo-de-35-milhoes-para-desenvolver-solucoes-em-braga-428915> (accessed on 27/08/2020).
- [29] H. Gholamhosseini S. M. Afifi and R. Sinha. 'Hardware Implementations of SVM on FPGA: A State-of-the-Art Review of Current Practice'. In: *International Journal of Innovative Science, Engineering & Technology (IJSET)* 2 (Nov. 2015).
- [30] V. Sahula R. Patil G. Gupta and A. Mandal. 'Power Aware Hardware Prototyping of Multiclass SVM Classifier Through Reconfiguration'. In: *25th International Conference on VLSI Design (VLSID)* (2012).
- [31] C. Kyrkou and T. Theocharides. 'SCoPE: towards a systolic array for SVM object detection'. In: *IEEE Embedded Systems Letters* 1 (Aug. 2009).
- [32] K. Benkrid H. Hussain and H. Seker. 'Novel Dynamic Partial Reconfiguration Implementations of the Support Vector Machine Classifier on FPGA'. In: *Turkish Journal of Electrical Engineering and Computer Sciences* 1 (Sept. 2014).
- [33] H. Corporaal and M. Arnold. 'Using Transport Triggered Architectures for Embedded Processor Design'. In: *Integr. Comput. Aided Eng.* 5 (1998), pp. 19–38.
- [34] K. Benkrid H. Hussain and H. Seker. 'Reconfiguration-Based Implementation of SVM Classifier on FPGA for Classifying Microarray Data'. In: *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (July 2013).
- [35] M. Papadonikolakis and C. Bouganis. 'Novel Cascade FPGA Accelerator for Support Vector Machines Classification'. In: *IEEE Transactions on Neural Networks and Learning Systems* 23 (July 2012).
- [36] A. H. M. Jallad and L. B. Mohammed. 'Hardware Support Vector Machine (SVM) for Satellite on-Board Applications'. In: *NASA/ESA Conference on Adaptive Hardware and Systems* (June 2014).

- [37] M. P. Sarma B. Mandal and K. K. Sarma. 'Implementation of Systolic Array Based SVM Classifier Using Multiplierless Kernel'. In: *International Conference on Signal Processing and Integrated Networks* (June 2014).
- [38] H. Khosravi D. Mahmoodi A. Soleimani and M. Taghizadeh. 'FPGA Simulation of Linear and Non-linear Support Vector Machine'. In: *Journal of Software Engineering and Applications* 4 (May 2011).
- [39] P. Yeyong M. Ning W. Shaojun and P. Yu. 'Implementation of LS-SVM with HLS on Zynq'. In: *International Conference on Field-Programmable Technology* (Dec. 2014).
- [40] T. Koide et al. 'FPGA Implementation Of Type Identifier For Colorectal Endoscopic Images With NBI Magnification'. In: *IEEE Asia Pacific Conference on Circuits and Systems* (Oct. 2014).
- [41] T. Koide et al. 'Customizable Hardware Architecture of Support Vector Machine in CAD System for Colorectal Endoscopic Images with NBI Magnification'. In: *18th Workshop on Synthesis and System Integration of Mixed Information Technologies* (June 2013).
- [42] Richard Wain et al. *An overview of FPGAs and FPGA programming-Initial experiences at Daresbury*. Tech. rep. Nov. 2006.
- [43] Xilinx. "Zynq-7000 SoC Data Sheet: Overview". Version 1.11.1. 2nd July 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (accessed on 14/10/2020).
- [44] "AMBA® AXI™ and ACE™ Protocol Specification". IHI 0022D. Revision E. ARM. Feb. 2013. URL: <https://developer.arm.com/documentation/ih0022/e/> (accessed on 14/10/2020).
- [45] Xilinx. "7 Series FPGAs Memory Resources". Version 1.14. 3rd June 2019. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (accessed on 14/10/2020).
- [46] Xilinx. "7 Series DSP48E1 Slice". Version 1.10. 27th Mar. 2018. URL: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf (accessed on 14/10/2020).
- [47] Digilent. "Zybo Z7 Board Reference Manual". Revision B. 21st Feb. 2018. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf (accessed on 14/10/2020).

- [48] "DACxx6x Dual 16-, 14-, 12-Bit, Low-Power, Buffered, Voltage-Output DACs With 2.5-V, 4-PPM/°C Internal Reference". SLAS719E. Revision E. Texas Instruments. June 2015. URL: <https://www.ti.com/lit/gpn/DAC8162> (accessed on 14/10/2020).
- [49] "ADS833x Low-Power, 16-Bit, 500-kSPS, 4- and 8-Channel Unipolar Input Analog-to-Digital Converters With Serial Interface". SBAS363E. Revision E. Texas Instruments. Aug. 2016. URL: <https://www.ti.com/lit/gpn/ADS8332> (accessed on 14/10/2020).
- [50] "ADT7301: $\pm 1^\circ\text{C}$ Accurate, 13-Bit, Digital Temperature Sensor". D02884-0-6/11. Revision B. Analog Devices. June 2011. URL: <https://www.analog.com/media/en/technical-documentation/data-sheets/ADT7301.pdf> (accessed on 14/10/2020).
- [51] "CP30H Series: 3.0A High Performance Peltier (TEC) Module". D02884-0-6/11. Version 1.03. CUI Devices. Oct. 2019. URL: <https://www.cuidevices.com/product/resource/pdf/cp30h.pdf> (accessed on 14/10/2020).
- [52] Xilinx. *Vivado Design Suite - HLx Editions*. URL: <https://www.xilinx.com/products/design-tools/vivado.html> (accessed on 14/10/2020).
- [53] Xilinx. *Vitis Unified Software Platform*. URL: <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html> (accessed on 14/10/2020).
- [54] "SPI Block Guide". S12SPIV4/D. Version 4.01. Motorola, Inc. June 2004.
- [55] James Rumbaugh, Ivar Jacobson and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.
- [56] Xilinx. "Vivado Design Suite User Guide". Version 2018.3. 19th Dec. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug901-vivado-synthesis.pdf (accessed on 14/10/2020).
- [57] Clifford E Cummings. 'Clock domain crossing (cdc) design & verification techniques using system-verilog'. In: (2008).
- [58] 'Arithmetic Operations: Division'. In: *Synthesis of Arithmetic Circuits*. John Wiley & Sons, Ltd, 2005. Chap. 6, pp. 109–163. ISBN: 9780471741428.
- [59] Gustavo Sutter et al. 'Power aware dividers in FPGA'. In: *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer. 2004, pp. 574–584.

- [60] J. Deschamps and G. Sutter. 'Decimal division: Algorithms and FPGA implementations'. In: *2010 VI Southern Programmable Logic Conference (SPL)*. Mar. 2010, pp. 67–72. DOI: 10.1109/SPL.2010.5483000.
- [61] Gustavo Sutter and Jean-Pierre Deschamps. 'High speed fixed point dividers for FPGAs'. In: *2009 International Conference on Field Programmable Logic and Applications*. IEEE. 2009, pp. 448–452.
- [62] Xilinx. "*Design Analysis and Closure Techniques*". Version 2017.3. 6th June 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug906-vivado-design-analysis.pdf (accessed on 15/11/2020).
- [63] V. Vapnik C. Cortes. 'Support-Vector Networks'. In: *Machine Learning* 20 (Sept. 1995).
- [64] M. Aizerman. 'Theoretical foundations of the potential function method in pattern recognition learning'. In: *Automation and Remote Control* 25 (June 1964).
- [65] Bernhard E. Boser, Isabelle M. Guyon and Vladimir N. Vapnik. 'A Training Algorithm for Optimal Margin Classifiers'. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Association for Computing Machinery, July 1992.
- [66] "J. Friedman". "'Another approach to polychotomous classification'". In: "*Technical Report, Statistics Department, Stanford University*" (1996).
- [67] Ulrich H.-G. Kreßel. 'Pairwise Classification and Support Vector Machines'. In: *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 255–268. ISBN: 0262194163.
- [68] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019). DOI: 10.1109/IEEESTD.2019.8766229.
- [69] Erick Oberstar. 'Fixed-point representation & fractional math (This is the Old Previous Copy See updated Link in Abstract)'. In: 1 (Jan. 2007).
- [70] Yuki Ago, Koji Nakano and Yasuaki Ito. 'A Classification Processor for a Support Vector Machine with Embedded DSP Slices and Block RAMs in the FPGA'. In: *Proceedings of the 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip*. MCSOC '13. IEEE Computer Society, 2013, pp. 91–96.

-
- [71] Davide Anguita, Andrea Boni and Sandro Ridella. 'Digital VLSI algorithms and architectures for Support Vector Machines'. In: *International journal of neural systems* 10.03 (2000), pp. 159–170.
- [72] D. Anguita et al. 'A Hardware-friendly Support Vector Machine for Embedded Automotive Applications'. In: *2007 International Joint Conference on Neural Networks*. 2007, pp. 1360–1364. DOI: 10.1109/IJCNN.2007.4371156.
- [73] Bosch Semiconductors. *Automotive Electronics - License Conditions, CAN Protocol and CAN FD Protocol*. URL: https://www.bosch-semiconductors.com/media/ip%5C_modules/pdf%5C_2/can%5C_protocol%5C_license%5C_conditions.pdf (accessed on 14/10/2020).