Methodology and architectures for
system-on-chip real-time
control

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

• A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy at Loughborough University.

Metadata Record: https://dspace.lboro.ac.uk/2134/34755

Version: Not specified

Publisher: © Dario Luis Sancho Pradel

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
University Library

Author/Filing Title ........................................... SANCHO  PROCEL  D.

Class Mark ...................................................... T

Please note that fines are charged on ALL overdue items.

FOR REFERENCE ONLY
Methodology and Architectures for System-on-Chip Real-Time Control

by

Dario Luis Sancho-Pradel

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of

Doctor of Philosophy

of

Loughborough University

June 2006

© Dario Luis Sancho Pradel
Abstract

Control systems are being embedded in an ever-increasing number of applications, many of them unfeasible prior to the joined advances in control, solid state technology and microelectronics design. In the era of information, communication and technology, society continuously absorbs new technologies and quickly demands new tools, services and commodities that require challenging control techniques operated at high sample rates. However, the implementation of modern control systems is by no means straightforward and requires multidisciplinary knowledge in fields of control, hardware and software design. Complex control laws with demanding real-time constraints are usually implemented using high-end devices combined with some custom logic in project-dependent, hence non-reusable solutions. Such high-end devices are intended to be flexible enough to cover a wide range of applications and therefore their architectural complexity often leads to inefficient control solutions in terms of power consumption, area and even performance. Furthermore, in order to fulfil the real-time requirements, complex device-dependent arithmetic routines have to be usually handcrafted in assembly language as most modern compilers are unable to make use of the narrow data vector instructions that are especially tailored for signal processing applications, and sometimes, the required performance can only be reached by a number of these devices running in parallel.

This thesis presents a comprehensive study of embedded real-time control implementations, bringing together hardware/software co-design, development tools and controller formulation aspects under a common framework that leads to a novel design methodology. Initially, standard and state-of-the-art control implementation techniques are reviewed, extracting the particular requirements found in most embedded real-time control applications. Then a novel fit-to-purpose reconfigurable control system processor (rCSP) soft-IP core is presented. It provides a low-cost, size and power consumption aware processing architecture particularly optimised for control applications with extra requirements in terms of sample rates, performance and connectivity.
In order to achieve numerical robustness in reduced word length oversampled real-time systems, a formulation based on the Delta-Transform is developed and used in the design of the processor's arithmetic unit and its associated development tools.

The second part of this thesis is engaged in the research of architectural and design alternatives for handling the increasing complexity of modern and future control systems. As a result, a scalable system-on-chip solution targeted for real-time embedded control is presented. This solution is architecturally efficient, technology independent and can readily interface both digital and analogue systems. Its multiprocessor design diverts all the real-time control related tasks to the rCSP while an inexpensive embedded CPU can easily coordinate the system's actions and periodically adapt the control coefficients.

Finally, one of the main problems found in the implementation of high-performance control systems is addressed, namely the embedded system programming. With the complexity of current designs continuously growing, the amount of architecture-dependent code required to implement not only the control law but also the interfacing among the different system's components and peripherals becomes an issue that leads to an extensive debugging process. Focussed on this problem, the research efforts resulted in a novel approach where most of the design-to-implementation steps are automated by a MATLAB-based tool that allows the design, simulation and automated system programming of rCSP-based control solutions from a block diagram representation of the controller. The thesis concludes with detailed control examples where its different research contributions are summarise.
Acknowledgements

I wish to thank my supervisor Professor Roger M. Goodall for his support and advice throughout this research, as well as Dr. Jose L. Nunez-Yanez and Dr. Vasilios Chouliaras for their technical expertise in digital systems design.

For my financial support I thank the UK Government's Engineering and Physical Science Research Council (EPSRC) for funding my research through their grant number GR/R38002.

I want to thank those people who have visited me ever since I left Spain, sharing with me experiences and adventures, and extending their friendship far from the geographical borders of my home country.

I feel indebted to my parents Adolfo and Soledad, and my brother Cesar for their unconditional love and patience, and to my new bulgarian family for accepting and loving me as a son of their own.

Finally, I deeply wish to thank Elena, my wife, for the support, advice, tenderness and care that she steadily provides me with throughout the years. It is for her that I try everyday to become not only a better scientist but a better person.
### Abbreviations and Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/D</td>
<td>Analogue-to-digital</td>
<td>IP</td>
<td>Intellectual property</td>
</tr>
<tr>
<td>ADC</td>
<td>Analogue-to-digital converter</td>
<td>ISR</td>
<td>Interrupt service routines</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-performance Bus</td>
<td>LC</td>
<td>Logic Cell (Xilinx)</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
<td>LE</td>
<td>Logic Element (Altera)</td>
</tr>
<tr>
<td>APB</td>
<td>Advanced Peripheral Bus</td>
<td>LTI</td>
<td>Linear time invariant</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application specific Integrated Circuit</td>
<td>LTV</td>
<td>Linear time variant</td>
</tr>
<tr>
<td>ASP</td>
<td>Application specific processor</td>
<td>LUT</td>
<td>Look-up Table</td>
</tr>
<tr>
<td>AU</td>
<td>Arithmetic Unit</td>
<td>MAC</td>
<td>Multiply-accumulate</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block (Xilinx)</td>
<td>MCU</td>
<td>Microcontroller unit</td>
</tr>
<tr>
<td>CPI</td>
<td>Clock cycles per instruction</td>
<td>PCB</td>
<td>Printed circuit board</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>PLD</td>
<td>Programmable Logic Device</td>
</tr>
<tr>
<td>D/A</td>
<td>Digital-to-analogue</td>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital-to-analogue converter</td>
<td>RTL</td>
<td>Register-transfer level</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access</td>
<td>SAC</td>
<td>Successive Approximation Converter</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
<td>SAR</td>
<td>Successive Approximation Register</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
<td>SIMD</td>
<td>Simple Instruction Multiple Data</td>
</tr>
<tr>
<td>ESA</td>
<td>European space agency</td>
<td>SoC</td>
<td>System-on-chip</td>
</tr>
<tr>
<td>ESB</td>
<td>Embedded System Block (Altera)</td>
<td>SRAM</td>
<td>Static Ram</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field programmable gate array</td>
<td>UART</td>
<td>Universal Asynch. Receiver Transmitter</td>
</tr>
<tr>
<td>FWL</td>
<td>Finite word length</td>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware description language</td>
<td>VHDL</td>
<td>VHSC Hardware Description Language</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$</td>
<td>Shift operator</td>
</tr>
<tr>
<td>$q^{-1}$</td>
<td>Delay operator</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Delta operator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>Z-Transform variable</td>
</tr>
<tr>
<td>$s$</td>
<td>Laplace Transform variable</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Delta-Transform variable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_l$</td>
<td>Exponent Length</td>
</tr>
<tr>
<td>$m_l$</td>
<td>Mantissa Length</td>
</tr>
<tr>
<td>$b_e$</td>
<td>Biased Exponent</td>
</tr>
</tbody>
</table>
Contents

List of Figures xii

List of Tables xviii

1 Introduction 1

1.1 Introduction 1

1.2 Motivation and State-of-the-Art 2

1.2.1 The gap between design and implementation 2

1.2.2 Inadequacy in the traditional formulation used in real-time systems 3

1.2.3 Inadequacies of off-the-shelf devices for control 5

1.3 New Technologies 7

1.3.1 System-on-chip solutions 7

1.3.2 FPGA technology 9

1.3.3 Compiling softwares to gates 10

1.4 Research Objectives 11

1.5 Summary of achievements 12

1.6 Structure of the thesis 13

2 Background on Digital Control I. Formulation 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Chapter objectives</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Continuous Control Fundamentals</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1 Brief history</td>
<td>18</td>
</tr>
<tr>
<td>2.2.2 Introduction</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Continuous Systems Description</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4 Controller Design</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Digital Control Fundamentals</td>
<td>22</td>
</tr>
<tr>
<td>2.3.1 Discrete System Description</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Adaptive Control</td>
<td>27</td>
</tr>
<tr>
<td>2.3.3 Intelligent Control</td>
<td>28</td>
</tr>
<tr>
<td>2.4 Digital Filtering and Signal Processing</td>
<td>29</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>31</td>
</tr>
<tr>
<td>3 Background on Digital Control II. Processor Architectures</td>
<td>32</td>
</tr>
<tr>
<td>3.1 Introduction and Objectives</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Implementation Requirements for Embedded Real-Time Digital Con-</td>
<td>33</td>
</tr>
<tr>
<td>trollers</td>
<td></td>
</tr>
<tr>
<td>3.2.1 Input/Output Requirements</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2 Arithmetic Requirements</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3 Other Requirements</td>
<td>38</td>
</tr>
<tr>
<td>3.2.4 FPGA-based designs</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Basic Concepts on Computer Architecture for Digital Signal Process-</td>
<td>39</td>
</tr>
<tr>
<td>ing</td>
<td></td>
</tr>
<tr>
<td>3.3.1 Instruction Set Architecture (ISA)</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2 Memory Hierarchy</td>
<td>45</td>
</tr>
</tbody>
</table>
3.3.3 Parallelism ............................................. 46

3.4 Operation-Level Parallelism.
Single Instruction Multiple Data (SIMD) .......................... 46

3.5 Instruction-Level Parallelism (ILP) ............................. 48

3.5.1 Single-Issue Architectures ................................. 48

3.5.2 Multi-Issue Architectures ................................. 49

3.6 Devices and Architectures for Digital Control .................. 52

3.6.1 General Purpose Embedded Micro Processor ............... 52

3.6.2 Digital Signal Processor (DSP) ............................ 53

3.6.3 Microcontroller Unit (MCU) ............................... 55

3.6.4 Application Specific Processor ............................ 56

3.7 Summary .................................................. 57

4 Delta Formulation and Floating-Point Implementation ............ 61

4.1 Introduction and Objectives of the Chapter ..................... 61

4.2 Delta-based State-Space Formulation .......................... 62

4.2.1 Canonic Delta Structure .................................. 62

4.2.2 Modified Canonic Delta Structure ........................ 66

4.2.3 Real-time Implementation ................................. 69

4.3 Reduced Word Length Floating-Point Implementations ........... 70

4.3.1 Floating-point formulation of the rCSP processor ........... 72

4.4 Summary .................................................. 74

5 Reconfigurable Control System Processor (rCSP) Architecture .... 76

5.1 Introduction and objectives of the chapter ..................... 76
5.2 Architecture I: Overview ........................................ 78
5.3 Architecture II: Instruction Set Architecture .............. 79
  5.3.1 Instruction Set and Instruction Encoding .................. 80
  5.3.2 Storage Type ............................................. 83
  5.3.3 Memory Addressing ........................................ 83
  5.3.4 Type and Size of Operands ................................ 85
  5.3.5 Operations in the Instruction Set ........................... 87
5.4 Implementation .................................................. 88
  5.4.1 Memory System ............................................ 92
  5.4.2 Arithmetic Unit ........................................... 95
  5.4.3 Pipelining and Datapath .................................. 96
  5.4.4 Internal Registers ....................................... 100
  5.4.5 I/O Modules ............................................. 102
5.5 AMBA External Interface and IL-MOD ......................... 107
  5.5.1 Software considerations .................................. 111
5.6 Reconfigurability ............................................... 111
5.7 Parallel Architectures ........................................... 113
5.8 Implementation Figures (Standard Configuration) ............ 115
5.9 Implementation Figures (Various Configurations) .......... 119
5.10 Summary ....................................................... 120

6 System-on-Chip Solution for Embedded Real-Time Control 123
  6.1 Introduction and Objectives of the Chapter .................. 123
  6.2 Architecture Overview ....................................... 124
6.2.1 Embedded CPU ..................................... 127
6.2.2 rCSP processor ................................... 128
6.2.3 Memory System .................................... 128
6.2.4 External Communication ............................ 129

6.3 LEON-Based System-on-Chip Implementation .................. 129
6.3.1 LEON Overview ................................... 130
6.3.2 Tailoring LEON for the system-on-chip control solution ...... 132
6.3.3 Final Remark ...................................... 134

6.4 Embedded Software ................................... 135
6.4.1 Architecture Overview ............................... 136
6.4.2 Functional Description ............................... 138
6.4.3 LEON-specific software issues ......................... 145

6.5 Monitoring and Commanding the System-on-Chip solution ...... 146

6.6 System-on-chip implementation figures ........................ 147

6.7 Summary ............................................. 150

7 rCSP-based Design Methodology and rCSP Development Suite 152
7.1 Introduction and Objectives of the Chapter .................... 152
7.2 Traditional Approach ................................... 153
7.3 Design-to-Implementation Methodology ........................ 154

7.4 rCSP Development Suite ................................ 157
7.4.1 Main Application Window ............................. 157
7.4.2 Memory Structure Module ............................. 163
7.4.3 Simulink's rCSP Models .............................. 164
4.2 Reallocation example of coefficients into the delta path .............. 66
4.3 Block diagram representation of $F(\gamma)$ in Modified Canonic Delta Form 67
4.4 IEEE single precision floating-point representation ...................... 71
4.5 Resolution of a 7-bit (rCSP) floating-point formulation for the interval $\nu \in [0, 1]$ ........................................ 72
4.6 Graphical representation of the resolution distribution for $\nu \in [0, 1]$ .... 73

5.1 rCSP General Instruction Encoding. OPC stands for opcode and Op stands for operand ................. 82
5.2 rCSP Instruction Set Encoding ........................................ 82
5.3 Digital representation of coefficients and variables. Ovf represents the overflow bits in the fixed-point representation. exp represents the exponent of the floating-point representation. .................. 86
5.4 Digital representation of the input values ............................... 87
5.5 rCSP’s internal design overview ....................................... 89
5.6 rCSP’s pinning overview ............................................... 91
5.7 rCSP Register-Register storage architecture ........................... 92
5.8 Dual port memory’s internal design .................................... 93
5.9 rCSP dataflow with MMU ............................................... 94
5.10 CFF-MOD internal architecture ....................................... 95
5.11 Arithmetic Unit port scheme .......................................... 96
5.12 Arithmetic Unit Pipelining ............................................ 96
5.13 Pipeline hazards due to instruction dependency. IM stands for Instruction Memory while DM stands for Data Memory .......................... 98
5.14 Data forwarding through different pipeline stages. DM stands for *Data Memory* .............................................. 99

5.15 Chain of gains leading to a pipeline hazard ........................................ 100

5.16 Pre and post-gain configuration that avoids the third pipeline hazard . 101

5.17 rCSP’s Internal registers ................................................................. 102

5.18 I-MOD architecture implementing SAR converters ......................... 104

5.19 I-MOD architecture optimised for digital inputs ............................. 105

5.20 Output block connected to both, digital and analogue ports .......... 106

5.21 Samples contribution to the PWM output, for a ratio of 3 ............... 107

5.22 PWM functional diagram and clock domains .................................. 107

5.23 rCSP’s internal states and transitions ............................................. 109

5.24 AMBA address word structure ....................................................... 109

5.25 Multi-rCSP architecture for faster algorithm execution. ................. 115

5.26 Memory distribution among different rCSP modules ....................... 116

5.27 Annotated RTL view of the rCSP’s core ........................................ 122

6.1 Block diagram of a complete embedded control system outlining the developed system-on-chip architecture ........................................... 125

6.2 SoC functional diagram ................................................................. 126

6.3 LEON2 architectural diagram [13] ................................................. 131

6.4 SoC implementation using LEON2 processor as embedded CPU ........ 134

6.5 Embedded software boot process flowchart ..................................... 139

6.6 rCSP’s boot sequence performed by the embedded CPU ................. 141

6.7 rCSP’s test sequence performed by the embedded CPU ................. 141
6.8 Commanding Loop structure ............................................. 142
6.9 Generic adaptation algorithm called periodically by the Timer's IRQ . 144
6.10 Action schedule within the Adaptation IRQ .......................... 145
6.11 Snapshot of the Java-based tool monitoring the controller and the sys-
etem status. ................................................................. 147
6.12 Annotated RTL view of the system-on-chip solutions ............... 151

7.1 Proposed design-to-implementation methodology ....................... 156
7.2 MIMO System Development ............................................. 158
7.3 Menu Bar unfolded ...................................................... 158
7.4 Arithmetic unit configuration window .................................. 159
7.5 MATLAB's representation of a MIMO subsystem ..................... 160
7.6 Block diagram representation of a 3 input 1 output MIMO system ... 162
7.7 An extract of the genControllerCff.m template ...................... 162
7.8 Memory Structure general window and coefficients' value display ...... 164
7.9 Simulink's rCSP models and coefficients modules .................... 165
7.10 rCSP Development Suite-based design flow ........................ 166
7.11 Main components of the system to be modelled ...................... 169
7.12 Analogue Controller Block Diagram .................................. 169
7.13 Project Configuration window ....................................... 170
7.14 genControllerCff.m tailored for the F14 Flight Controller application 171
7.15 Generated assembly code, machine code and memory structure ...... 172
7.16 rCSP Development Suite's main module showing the generated real-
time equations ............................................................ 173
7.17 F-14 flight control Simulink model using the rCSP as digital controller 174

7.18 rCSP outputs selection ........................................... 175

7.19 Airplane’s response to the pilot’s commands using the original analogue controller and the soft model of the rCSP with MATLAB’s floating-point arithmetic ............................................. 175

7.20 Airplane’s response using rCSP’s soft model with rounded off coefficients 176

7.21 Airplane’s response for a variety of coefficient configurations .... 176

7.22 Command vs Actual Angle of Attack using the actual rCSP’s hardware model .......................................................... 177

7.23 Extract from the module rcspv30_boot.c .................................. 178

7.24 Block diagram description of (a) HDD’s model (b) Controller .... 180

7.25 HDD control system implemented in the rCSP .......................... 181

7.26 Controller description in the rCSP Development Suite .......... 182

7.27 genControllerCff.m tailoring (2MIMO2DEln syntax) ................ 183

7.28 genControllerCff.m tailoring (MIMO2ln syntax) ...................... 184

7.29 Simulink test bench of the HDD control system ...................... 185

7.30 System’s response comparison for validating the real-time equations .. 186

7.31 System’s response with quantised (delta) coefficients ................. 186

7.32 Analogue and delta-formulated digital controller (\( m_l = 6 \)) comparison in presence of measurement noise ........................................ 187

7.33 System’s response with quantised (z-based) coefficients ............ 187

7.34 System’s response with the rCSP’s actual hardware model (noiseless) 188

7.35 System’s response with the rCSP’s actual hardware model (including Gaussian measurement noise) ........................................ 189
7.36 System’s response (z formulation) with the rCSP’s actual hardware model (noiseless) ................................................. 190

7.37 Extract from the VHDL module generated by the rCSP Development Suite ................................................................. 191

7.38 Client-Server architecture for remote bitstream generation .......... 192

A.1 Excalibur Embedded Processor PLD Architecture [2] ................. 201
A.2 Basic Spartan-IIE Family FPGA Block Diagram [49] .............. 202
A.3 Virtex-II Architecture Overview [51] ................................. 204

D.1 Application Launcher .................................................... 217

E.1 Internal structure of $\delta^{-1}$ operator ............................... 220

F.1 Common discretising routes to obtain $F(\gamma)$ ..................... 224
## List of Tables

3.1 Most commonly used addressing modes in general purpose processors . 41

3.2 Categories of instruction operators ........................................... 43

4.1 Coefficients’ parameters in the rCSP’s standard configuration .... 72

5.1 Summary of digital control requirements covered by the rCSP processor 79

5.2 rCSP Instruction Set, meaning and usage .............................. 81

5.3 Addressing modes supported by rCSP ....................................... 84

5.4 rCSP’s internal data memory map ........................................... 85

5.5 Operands types supported by the rCSP ................................. 86

5.6 Categories of instruction operators ....................................... 88

5.7 rCSP pipeline ................................................................. 98

5.8 rCSP’s internal registers’ description .................................... 101

5.9 Status registers description ................................................ 102

5.10 Error State description .................................................... 103

5.11 High-level commands accepted by the IL-MOD ...................... 108

5.12 AMBA address partitioning ............................................... 108

5.13 List of memory mapped modules and their corresponding AMBA base addresses ........................................... 110
7.3 Absolute (peak) position-error for a number of coefficient configurations

7.4 System Parameters. HDD's values except for $b$ are courtesy of Hitachi Corporation, Japan

7.5 Implementations' arithmetic word length requirements (in bits)

A.1 Targeted FPGAs

A.2 Excalibur FPGA main complexity parameters

A.3 Selected Spartan-IIIE FPGAs main complexity parameters

A.4 Virtex2 FPGA main complexity parameters

B.1 rCSP processor VHDL model hierarchy

B.2 Packages used in the rCSP processor model

D.1 tfz2tfdel function's parameters list

D.2 Function tfdel2ssmde1 parameters list

D.3 Function tfs2ssmde1 parameters list

D.4 Function ssz2tfdel parameters list

D.5 Function tfdel2ssmde1 parameters list

D.6 Function cff2num parameters list

D.7 Function quantcff parameters list

D.8 Function num2stv parameters list

D.9 Function stv2num parameters list
Chapter 1

Introduction

1.1 Introduction

Systems whose performance or characteristics are enhanced by the actions of a controller are becoming more and more often part of our daily commodities. The automotive industry provides good examples of everyday systems that would suffer a serious loss in functionality and performance if they were not supported by embedded controllers in a number of tasks like servo-steering, ABS braking, front light adjustment, and active suspension amongst many. In most cases, these controllers must be embedded in the controlled system and implemented using digital devices that should balance performance and resources (e.g. cost, size, power consumption), both being tightly related to the advances in solid state technology and electronic design. The continuous progress in these areas allows the implementation of control systems for applications that were not feasible before. A good example of the relationship between control theory and digital design is the emerging field of mechatronics where only by mixing control, electronic systems and mechanical systems can new functionalities be achieved [7]. Some examples of these new functionalities and systems are: drive-by-wire, brake-by-wire and active suspension in the automotive industry, the new blended wing design aircraft and fly-by-wire in the aeronautic industry, and ROVs (remotely operated vehicles) in underwater exploration.
In order to achieve the performance and functionality required by many of these challenging new applications, modern control systems must be able to:

- Operate with hard real-time constraints.
- Be embedded in the systems to be controlled.
- Operate at high-sample rates.
- Seamlessly interface with other digital and analogue devices.
- Execute other functions apart from the control law.
- Be upgradable.
- Be power consumption aware.

Apart from these characteristics, other important requirements common to most commercial products are robustness, cost and time-to-market.

1.2 Motivation and State-of-the-Art

1.2.1 The gap between design and implementation

Embedded real-time control solutions are the key to many of the newest advances in a wide variety of fields. However, their implementation is far from straightforward and usually requires multidisciplinary knowledge in the fields of control, hardware and software design. In contrast, many control engineers want their control law to be implemented as straightforward as possible, and generally they have neither the skill nor the will to achieve efficient, cost-effective solutions [17]. Traditionally, the lack of rigorous research in real-time control implementations ([31], [48]) led to approaches where the aspects of control and electronic design were disjointed. The increased complexity of control systems shows a number of inadequacies in current solutions [32], which are typically based upon off-the-shelf devices such as digital signal processors (DSPs), microcontroller units (MCUs) and general-purpose processors (sometimes embedded in
parallel architectures). As the computational complexity and speed required for modern control systems inexorably rise, in a number of cases the computationally intensive nature of control law execution will make a general-purpose device unable to perform all the calculations within the time frame of the desired sample period, enforcing a lower sample rate and leading to numerical and stability problems. Moreover, contemporary control solutions require much more than just control law execution. They need signal processing, standard communication channels, more sophisticated control techniques, etc. Scalable architectures that allow an easy integration, upgrade and/or replacement are required as time to market is nowadays one of the biggest constraints for developers and the verification process of a new or re-engineered product is probably the most time and budget consuming. All these features and functionalities require extra computational efforts, and often some specialised hardware architectures must be included at extra cost in terms of resources and added complexity in the design, implementation and programming of the system. However, recent developments in FPGA technology offer very attractive and efficient solutions for a variety of applications, including embedded control. Nevertheless traditional control implementations are based on off-the-shelf devices and don't take advantage of the possibilities this new technology offers. In general, current improvements in semiconductor technology have led to implementations based on faster and more general devices with a dense gate distribution, relying mainly on the increasing clock frequencies, putting aside once more the architectural requirements of control systems, thus increasing the gap between real-time control design and its effective high-performance implementation.

1.2.2 Inadequacy in the traditional formulation used in real-time systems

The potential errors that finite-word length arithmetic can introduce into the design (e.g. coefficient quantisation error, overflow error, roundoff error, etc.) are well known [26] as well as its benefits over floating-point architectures (e.g. speed, cost, size, power consumption). Such numerical precision problems depend not only on the hardware device but also on the mathematical approach used to model the system. Also well known are the numerical issues that arise by the use of the forward shift operator \((q)\)
and its related $Z$-Transform variable ($z$) combined with finite-word length arithmetic at high sample rates, although surprisingly this approach is nevertheless commonly used in many cases. The reason is probably that in the early days of implementing non-recursive filters discrete digital circuit elements were used, in which successive values of sampled variables were clocked through a bank of shift registers \cite{15}. Another reason is the equivalence between the inverse of the $z$ variable ($z^{-1}$) and a sample period delay, which provides an intuitive representation of digital synchronous systems. Ironically, the alternative discrete-time operator described next provides an equivalence with the Laplace operator and the transfer functions that are being emulated, but this fact is not widely appreciated.

An alternative formulation based on the $\delta$ operator

Very high sampling frequencies invariably result in long word lengths for both coefficients and variables within the controller, primarily because the difference between the successive values of the input and output become increasingly small. The problems associated with the design of controllers based on the shift operator are well known \cite{36}. Middleton and Goodwin \cite{39} have demonstrated the numerical benefits of the $\delta$ operator, presenting a unified approach between the formulation of continuous and discrete systems in which the delta operator is expressed as $\delta = (q - 1)/T$, having the property that $\delta \to s$ as $T \to 0$. Goodall and Donaghue \cite{18} presented a study of the implementation of very high sample rate digital filters using a small variation of the delta operator $\delta = q - 1$, which is focussed upon implementation's practicalities (further explained in Section 2.3.1 and Chapter 4). In their study, which includes a successful implementation on a DSP, they use a modified canonic form of the delta operator that places its recursive multipliers on the forward path, leading to a conformity of scaling for the filter's internal variables. The work showed that very high speed sampling (4 orders of magnitude higher than the bandwidth) is possible with a suitable numerical formulation.

Since this thesis is focussed upon real-time control applications with high requirements in terms of sampling frequency and controller complexity, the hardware architectures and mathematical tools developed are based on the $\delta$ operator formulation.
1.2.3 Inadequacies of off-the-shelf devices for control

In many cases, the performance required in hard constrained real-time control applications forces the controller to be: embedded (i.e. size aware), predictive (i.e. real-time aware), able to achieve high-sample rates (i.e. fast arithmetic execution), power aware, upgradable (by firmware or bitstream), and easily interfaced with other units. These requirements often overwhelm many designs based on off-the-shelf components for various reasons:

- High-performance off-the-shelf devices are expensive and/or power hungry. Examples can be found in most 32-bit/64-bit general purpose processors and high-end DSPs.

- Inexpensive devices targeted for very low power consumption and control normally don't match the computational requirements of high-performance advanced control systems.

- Most microcontrollers are based on simplified RISC architectures that keep an optimised subset of instructions. However, they are designed to support a wide range of applications and therefore a single application often uses only a small subset of the total resources available.

Section 3.6 reviews the main architectural characteristics of current off-the-shelf devices used for control.

In the early 1980s, three main architectures were adopted by most control engineers for implementing controllers and digital filters: microprocessors, microcontrollers and digital signal processors (DSPs). Clear representatives of those times were Zilog's Z80 8-bit microprocessor, Intel's 8051 8-bit microcontroller and Texas Instrument's TMS32010 16-bit DSP. Numerical routines were coded in assembly and optimised for 8-bit arithmetic (or 16-bit in the case of DSPs). However, at that time the performance requirements were more modest and the sample rates were relatively low. This scenario reduced the numerical requirements in terms of word length and dynamic range, and compensated the numerical issues introduced by use of the $z^{-1}$ variable. It is interesting to notice how many designs are nowadays still based on the same or a
modified version of some of these early 1980s devices. In fact, current control design techniques and implementations have not changed that much. What have changed greatly are the real-time performance and requirements such as power consumption, size and connectivity that modern control systems must fulfill. Although the devices currently used for implementing the control law have evolved their design through several generations, increased their speed (in some cases by several orders of magnitude), and improved their functionality, they have also widened their scope by trying to be valid solutions for a wide range of applications.

Application Specific Processors for control - An alternative

Despite the clear advantages that tailored architectures can provide for real-time embedded control applications, current research trends are mainly focused on the development of new generations of DSPs. In contrast, little work has been done in the development of targeted architectures able to address real-time control application that require high sample rates. This work will be summarise next. Jones et al [32] studied the hardware requirements for the design of a control system processor (CSP) for linear time invariant (LTI) systems, based on the modified delta canonic structure. They presented a MAC architecture suitable for this numerical formulation and outlined three different approaches for the control law execution (single MAC, parallel MACs, systolic array). Goodall et al [19] described the use of a non-volatile FPGA for implementing LTI controllers. They presented the hardware architecture of the first CSP (see Figure 1.1) based on the single MAC structure described in [32], wrapped with the necessary logic to execute the control law endlessly as a program stored on a ROM.

This thesis provides a comprehensive study into real-time embedded control implementations, and therefore it covers a wider range of aspects, from system-on-chip design to controller formulations. At the core of the system-on-chip architecture lies a new control system processor developed in this research, which is highly optimised for both stand-alone and system-on-chip execution, and capable of interfacing directly with analogue and digital stages. The initial version of this processor (CSP II) [46],[45] implemented an optimised architecture for real-time control and included a
limited software tool support capable of compiling control equations into its native machine-code language. However, this thesis will present the final reconfigurable version (rCSP v3.0) of the control processor (see Chapter 5), which allows its architecture to be moulded to the actual requirements of the application, avoiding the waste of resources and giving increased performance. Its design is vendor-independent, synthesises for both FPGA and ASIC technologies and its programming and simulation is fully supported and automated by its companion MATLAB-based software tools (see Chapter 7). Throughout the rest of the thesis, the version of the processor will be omitted being therefore referred as rCSP.

![CSP internal architecture](image)

Figure 1.1: CSP internal architecture

### 1.3 New Technologies

#### 1.3.1 System-on-chip solutions

An important trend in embedded systems is the use of processor cores together with application-specific circuitry. Often the application's functional and performance requirements are met by combining custom hardware solutions together with software running on a number of standard embedded processor cores, which are able to interface to such special-purpose hardware. In practice, embedded problems are usually solved by one of the three approaches [22]:

1. **SoC solution.** The designer uses a combined hardware/software solution that in-
 Chapter 1: Introduction

includes some custom hardware and an embedded processor core that is integrated with the custom hardware, often on the same chip.

2. *General purpose embedded processor.* The designer uses custom software running on an off-the-shelf embedded processor.

3. *DSP solution.* The designer uses a digital signal processor and custom software for the processor.

Due to the particular challenges in terms of performance requirements, resource constraints and safety that modern real-time embedded control systems present, system-on-chip solutions are envisioned to play a key role in current and future complex yet highly-integrated control designs. In order to address the complexity of distributed systems programming, solutions based on architecture description languages (ADLs) have been proposed. For instance, [23] presents a programming language (Giotto) that aims at distributed hard real-time applications with periodic behaviour, such as control systems of possibly distributed sensors, actuators, CPUs, and networks. This approach, however, forces control engineers to design complex and project-specific hardware architectures that will be then programmed and interfaced by means of a non-standard programming language. One of the most significative contributions to field of customisable system-on-chip architectures for control was recently released (2003) by IBM: The *Customizable Control Processor* (CCP) [24]. The CCP (see Figure 1.2) combines a prehardened superstructure on the chip with an area that can be customised for a particular application using standard cell logic and cores. The hardened superstructure is based on a PowerPC 405 or 440 processor and includes all basic peripheral cores common to many SoC designs. In the customisable area, a combination of customer logic, IP, and embedded memory can be connected to the CoreConnect bus structure and customer-defined Input/Output.

However, this solution comes with one third of its area predefined, includes a high-end PowerPC microprocessor and many different communication peripherals with the aim of being suitable for a wide range of application, such as hard disk controllers, voice over IP (VoIP) or game consoles. Therefore such a solution is substantially different from the approach proposed in this thesis, which takes full advantage of the
programmable nature of modern devices such as FPGAs, and creates an optimised, fit-for-purpose solution that maximises performance while decreasing the design gate count, hence reducing power consumption and implementation costs.

1.3.2 FPGA technology

Although the different components of the system-on-chip solution presented in this thesis are technology-independent (i.e. can be synthesised in either ASIC or FPGA technologies), there are certain advantages that make an FPGA-based implementation especially attractive:

- FPGA technology allows straight-forward and in-situ architectural upgrades, thus taking full advantage of the system's reconfigurability. In that way for example, an embedded controller could be updated periodically (e.g. due to plant changes, model updates, etc.) without having to replace its hardware.

- Current FPGAs include dedicated arithmetic blocks such as Multipliers and MAC units in their internal structure, thus improving the implementation performance of signal processing based designs.

- FPGA devices allow fast prototyping and prove to be cost-effective solutions in many applications.
• The fit-to-purpose nature of the design reduces the FPGA requirements in terms of size and speed, hence its implementation can be done using low-budget FPGAs instead of state-of-the-art devices.

• The design can benefit from the continuous improvements in the solid-state technology.

1.3.3 Compiling softwares to gates

The need for tailored architectures in control is widely recognised but hardware design skills are uncommon among control designers. Hardware Description Languages (HDLs) provide many hardware-centric features that make them superior to software programming languages, such as the ability to directly manipulate signals of any arbitrary length or the notion of actual concurrency and time. However, the long development and testing time inherent in the use of HDLs very often make off-the-shelf devices the only feasible option to meet time-to-market deadlines. This problem is recently attracting the attention of EDA design companies that try to provide tools for transforming the higher-level model of a system directly into RTL (register-transfer level) or gates. Today some tools can synthesise directly from C-based languages. However, synthesis of C/C++ or SystemC is much restricted in terms of language coverage and vendor support [34], and the performance, size and architecture of the final RTL is by no means deterministic. In general C/C++/SystemC are still mainly used for modelling, being therefore complementary to and not a replacement for traditional HDLs. In the field of digital signal processing, another option is to use Simulink to synthesise digital filters into an FPGA. This approach typically allows the rapid prototyping and implementation of static structures that lack flexibility and reprogrammability.

This thesis shares with Simulink the idea of automated controller programming directly from block-diagrams representation but presents a flexible processing structure (rCSP) that can be reprogrammed in real-time, being especially tailored to cooperate with other processing cores. On top of that structure, the thesis builds a whole system-on-chip architecture that provides a high-performance reprogrammable solution that features complex adaptation algorithms at high-sample rates.
1.4 Research Objectives

Digital design is a field in continuous evolution, not only driven by the periodic improvements in solid-state technology, but also due to the increasing functionalities and performance required by a wide variety of applications in fields like consumer electronics, personal computing or digital control and signal processing. The demands in these fields for faster, smaller and higher performing devices lead to the design of new digital architectures and supporting software systems.

However, as opposed to other fields, embedded real-time control is an interdisciplinary field often regarded to require a certain degree of “craftsmanship”, and where traditional approaches are still common in most real case applications. The most remarkable change is maybe the use of devices based on faster silicon technologies and hardware architectures that feature specialised arithmetic units and allow parallelism in their execution. However, compiler technology is not yet able to take full advantage of those new characteristics and very often the arithmetic-intensive parts of the algorithm have to be hand-coded in assembly language. As result, designers rely on faster devices and floating point arithmetic to achieve what could be done with only a fraction of the utilised resources and power consumption by a targeted architecture and a suitable numerical representation. Software generation for the selected implementation architecture is still an important, application-dependent and error-prone part of the design process.

The aim of this thesis is to provide a holistic approach to the study and implementation of embedded real-time control systems, demonstrating how, by bringing together embedded real-time control, automated software generation and microelectronic design under a common framework, the gap between the mathematical description of the controller and its high-performance final implementation can be removed. In order to do so, current design processes, hardware architectures and numerical representations are reviewed, and their strengths, shortcomings and trade-offs are analysed. This analysis provides the foundations for developing a reconfigurable fit-to-purpose processor (rCSP), and a system-on-chip targeted architecture focused exclusively on the real-time embedded control problem. Such an architecture and processor should be
designed with future control requirements in mind, be low cost, high-performing, easily programmable and upgradable. In order to achieve a straightforward implementation of digital controllers, different software-related issues should be addressed, especially the complex, error-prone and architecture-dependent software that control engineers are currently bound to produce in every project. It is clear that minimising or even eliminating software development tasks from the design flow of a controller would not only reduce time-to-market, but will minimise maintenance costs and will probably deploy better performance. This analysis leads to one of the main objectives of the thesis: to investigate how much of the design-to-implementation load can be taken away from the control designer by a suitable software tool made to support the development of rCSP-based control systems. Thus, the motivation of this work is not an out-and-out optimisation of existing architectures but the provision of a more-than-sufficient novel architecture combined with a degree of design automation that will allow practitioners to use the technology successfully.

Field Programmable Gate Array (FPGA) technologies are becoming increasingly important in industry, delivering performance and features that previously only an application specific integrated circuit (ASIC) device could provide. Their prices are becoming more and more competitive, and their design tools are often much more user friendly than their ASIC counterparts. They not only allow for fast prototyping, but they provide designs with a new functionality: reconfigurability. However, it has been suggested that FPGAs are still only a first-generation embodiment of the big idea of a general-purpose, reconfigurable substrate for special purpose computing [42]. In any case their advantages are important enough to investigate the role that this new technology can play in current and future control implementations.

1.5 Summary of achievements

This thesis addresses the problem of high-performance real-time embedded control implementations with demanding resource constraints, sample rates requirements and complexity. The original contributions of this thesis are summarised next.
Chapter 1: Introduction

1. Detailed study of \( \delta \)-operator based real-time control systems with special attention to the case of short word length floating-point implementations.

2. Design and development of a novel reconfigurable control system processor (rCSP) capable of satisfying the technological and functional targets of demanding real-time control systems. It is designed as a reconfigurable soft IP core that can be easily integrated as a building block in complex control architectures and tailored for each particular application, optimising in that way its implementation and reducing its power consumption.

3. Design and development of a novel novel multi-processor system-on-chip architecture targeted for advanced embedded real-time control solutions, which is based on the concurrent operation of an embedded processor and the rCSP. Its design addresses the functionalities of modern adaptive control systems, their device interfacing and communication issues, and its scalable architecture ensures the suitability of the design for future control scenarios.

4. Design and development of a software suite (rCSP Development Suite) that supports and automates the design-to-implementation stages of rCSP-based control solutions. Its novel design flow and implementation methodology minimises control engineers' hardware/software co-design efforts, thus allowing them to focus on the control algorithm’s development, reducing the implementation time and obtaining the highest performances.

1.6 Structure of the thesis

This thesis is structured as follows:

Chapter 1 puts this research into context, introduces the research objectives and motivations, and summarises its contributions.

Chapter 2 provides a background knowledge and literature survey in control systems, and their digital representation that will be used as a starting point to explore the implementation requirements that drive the architectures presented in Chapter
3 and that influence the design of the rCSP processor. This chapter summarises
the mathematical description of linear-time-invariant (LTI) and linear-time-variant
(LTV) systems, it outlines the different numerical operators used in the field of con­
trol and provides an introduction to real-time system. Some mathematical concepts
reviewed in this chapter are further explained in Appendices E and F. The δ operator
formulation, central to this thesis, is explained in detail in Chapter 4.

Chapter 3 concentrates on the actual implementation issues of digital control
systems. It starts describing the requirements that are particular to modern control
and signal processing algorithms. Then it provides an introduction to the state-of­
the-art computer architecture for signal processing in order to better understand the
different design options that can best match these requirements. Major embedded
architectures used for implementing digital control and signal processing algorithms
are reviewed, identifying their advantages and drawbacks, especially when operated at
high sample rates under embedded real-time requirements.

Chapter 4 presents an alternative state space representation of digital control sys­
tems based on the δ-operator (rather than the commonly used delay operator), whose
numerical robustness and well scaled coefficients allow to address high-performance
systems operated at high sample sample rates with short word length floating-point im­
plementations (i.e. significantly shorter than the IEEE 754 standard). Based on this
result, this chapter claims the feasibility of a real-time control-oriented arithmetic
unit design with reduced size and power consumption, yet high in performance. This
arithmetic unit is developed in Chapter 5 and implemented in the rCSP. A quanti­
tative comparison of the numerical properties and word length requirements between
the shift and delta operator is provided in Chapter 7.

Chapter 5 presents an inside view of the rCSP processor, describing in detail the
its architecture and implementation. This chapter also provides a detailed explanation
of how the rCSP interfaces with external devices such as embedded CPUs, sensors and
actuators. The chapter finishes describing briefly the possibility of building parallel
multi-rCSP architectures and providing rCSP's performance and post place & route
implementation figures.
Chapter 6 redescribes the embedded real-time control problem as a multiprocessor system-on-chip reconfigurable solution with automated system programming. This chapter presents the developed system-on-chip architecture as well as its implementation (as a demonstrator) based on the open-source SPARC-compatible LEON processor, whose development is supported by the European Space Agency (ESA). The particularities of the interfacing between this embedded CPU architecture and the rCSP are explained as well as the the embedded software architecture. The implementation figures for the developed system-on-chip solution are provided at the end of the chapter.

Chapter 7 covers the methodology and design flow developed in this thesis for the design and implementation of high-performance embedded real-time control systems. Its objective is to eliminate the gap between the controller design and its actual (high-performance) implementation, automating the system’s programming, hardware configuration and control system’s simulation. This is achieved by means of the rCSP Development Suite (described in detail in this chapter), a MATLAB tool developed to support multi-rCSP-based control systems. Finally, the design, simulation and implementation of a F-14 digital flight controller and a hard-disk-drive controller featuring Kalman filtering are presented to exemplify the design process described in the chapter.

Chapter 8 concludes the thesis by summarising the main findings and how they fulfil the objectives of the research.

Appendix A provides an overview of the main architectural characteristics of the FPGA devices used in the implementation of the rCSP processor and the system-on-chip solution. Appendix B presents the VHDL model hierarchy of the rCSP processor. Appendix C summarises the main device driver and control libraries developed for the system-on-chip solution. Appendix D presents a number of key MATLAB functions developed in the thesis to operate with the delta-transform and to generate and analyse coefficients and state variables for any arbitrary rCSP configuration. Appendix E explains in detail the different mathematical operators used in this thesis and the relation among them. Appendix F provides a background review on the different transform methods used in this thesis, presents and demonstrates some useful
properties of delta-transform systems. Appendix G provides the list of publications produced as result of this research.
Chapter 2

Background on Digital Control I.

Formulation

2.1 Chapter objectives

This chapter provides a background knowledge in control systems, and their digital representation that will be used as a starting point to explore the implementation requirements that drive the different architectures presented in Chapter 3 and that influence the design of the rCSP processor. The objectives of the chapter are to:

- briefly present the common mathematical formulation used to describe linear time invariant (LTI) systems and linear time variant (LTV) adaptive systems.
- review the concept of adaptation in control systems.
- introduce an alternative numerical formulation based on the delta operator ($\delta$), as opposed to the forward shift operator ($q$), especially suitable for high-sample rate systems.
- introduce the real-time concept for digital control systems and how to extract the real-time equations from the system description.
2.2 Continuous Control Fundamentals

2.2.1 Brief history

Control history goes back to more than 2000 years ago when Greeks and Arabs used water clocks to keep accurate track of time around the 3rd century BC. The Industrial Revolution in Europe was marked by the invention of power-driven machinery such as advanced grain mills, furnaces, boilers, and the steam engine (J. Watt, 1769). These devices could not be adequately regulated by hand, and so arose a new requirement for automatic control systems. A variety of control devices was invented, including float regulators, temperature regulators, pressure regulators, and speed control devices. However, the design of such devices was based mainly on intuition and experience. It was not until J.C. Maxwell provided the first rigorous mathematical analysis of a feedback control system (James Watt’s Fly ball governor) in 1868 that control theory started being a rigorous science.

2.2.2 Introduction

In general, a control system can be described by the combination of four basic elements, namely the system to be controlled (normally called plant, system or process), the controller, the sensors and the actuators. Figure 2.1 shows a basic feedback control diagram.

![Basic feedback control diagram](image)

Figure 2.1: Basic continuous feedback control scheme

Typically, the controller is calculated for minimising (and ideally canceling) the error between the desired output (encoded in the given input) and the actual feedback system output while accommodating its transient response (i.e. settling time, percent overshoot, etc.) and ensuring stability.
In order to design a controller the system to be controlled must be described mathematically (unless applying soft-computing methods as will be mentioned in Section 2.3.3). In general, a system belongs to one of the following groups:

1. General case: \( \dot{x}(t) = f(x, u, t) \)
2. Invariant system: \( \dot{x}(t) = f(x, u) \)
3. Linear Time-Variant (LTV): \( \dot{x}(t) = A(t) \cdot x(t) + B(t) \cdot u(t) \)
4. Linear Time-Invariant (LTI): \( \dot{x}(t) = A \cdot x(t) + B \cdot u(t) \)

This section and the following one will briefly show the most common mathematical descriptions of systems, giving special attention to LTI systems. In most of the cases, the plant cannot in general be described linearly, but it can be linearised around its normal operating point, obtaining an LTI model of the system valid in the proximity of this point. Sometimes, the plant has partially unknown dynamics and/or time-varying parameters but behaves in a linear fashion and can be described as an LTV system. Adaptive control schemes are widely used in these cases.

Although this research deals with the digital processing involved in the field of embedded real-time control, normally the system to be controlled is continuous and in many cases, before obtaining its final digital version, the controller is designed in the continuous domain. Other times there is already an analogue controller for a particular system that needs to be digitally implemented. Therefore, it is important to understand the mathematical formulation of continuous systems before reviewing digital implementations.

2.2.3 Continuous Systems Description

There are several ways to model continuous LTI systems mathematically. One way is to use linear differential equations with constant coefficients. A second method is to use transfer functions. A third type of model is the state-space model [9].
Chapter 2: Background on Digital Control I. Controller Formulation

Differential Equations

Physical systems are often mathematically modelled by a set of differential equations extracted from physical laws that under certain simplifying assumptions govern the dynamics of the system. The general form of these equations is:

\[ y^{(n)} + a_{n-1}y^{(n-1)} + \ldots + a_1y^1 + a_0y = b_m u^{(m)} + b_{m-1} u^{(m-1)} + \ldots + b_1 u^1 + b_0 u \]  \hspace{1cm} (2.1)

where \( y^{(n)} \) represents the \( n \)-th derivative of \( y \) with respect to time (i.e. \( \frac{d^n y}{dt^n} \)).

As it can be seen, Equation 2.1 represents an \( n \)-th order differential equation with constant coefficients (a special case of differential equations) and the systems governed by such equations are called Linear Time Invariant (LTI) systems. Although for small order equations analytical solutions can be easily obtained, higher order equations can be very difficult (or impossible) to work with in that way.

Transfer function

The solution of differential equations can be more easily found by means of the Laplace transform. The Laplace transform has the property of transforming linear differential equations into algebraic ones, much more suitable for further processing and analysis. Appendix F gives a more detailed description of different mathematical transforms frequently used in the study of control systems.

By applying the Laplace transform in both sides of Equation 2.1 (assuming zero initial conditions), and obtaining the ratio \( Y(s)/U(s) \) the following transfer function is obtained:

\[ F(s) = \frac{Y(s)}{U(s)} = \frac{b_m s^m + b_{m-1} s^{(m-1)} + \ldots + b_1 s + b_0}{s^n + a_{n-1} s^{(n-1)} + \ldots + a_1 s + a_0} \]  \hspace{1cm} (2.2)

A transfer function shows the relation between the input and the output of the different single-input single-output (SISO) elements that comprise an engineering system [9]. These are often represented in form of block diagrams.

The components of a block diagram for linear-time-invariant (LTI) system are shown in Figure 2.2 [40]. Notice in Figure 2.2(b) that \( C(s) = G(s) \cdot R(s) \). These components will be used in Chapter 7 to provide the rCSP software tools with a description of the controller to be implemented.
State-Space Models

The state-space technique is particularly useful for modeling complex multiple-input multiple-output (MIMO) engineering systems. Its matrix representation makes it especially suitable for digital processing. The basis of the state-space technique is the representation of a system by means of first-order coupled differential equations, known as state equations. The number of first-order differential equations required to model a systems defines the order of the system. Associated with the first-order differential equations are a set of state variables, the same number as there are differential equations. Equations 2.3 and 2.4 show a general representation of the state equations of a LTI system.

\[
\begin{align*}
\dot{x}(t) &= A \cdot x(t) + B \cdot u(t) \\
y(t) &= C \cdot x(t) + D \cdot u(t)
\end{align*}
\]

(2.3) (2.4)

For a system with \( n \) state variables, \( r \) inputs and \( p \) outputs:

- \( x = n \)-vector, known as the state vector.
- \( y = p \)-vector, known as the output vector.
- \( u = r \)-vector, known as the input vector.
- \( A = (n \times n) \) matrix, known as the state matrix.
- \( B = (n \times r) \) matrix, known as the input matrix.
- \( C = (p \times n) \) matrix, known as the output matrix.
- \( D = (p \times r) \) matrix, known as the direct transmission matrix.
The choice of the set of state variables is not unique leading to different representations (in fact there are infinitely many state-space representations of the same system). However, there is a number of especially useful representations called *canonic forms* and Equations 2.3 and 2.4 can be transformed into them by means of appropriate *similarity transformation* matrices (for more information on this subject consult for example [41] and [40]).

### 2.2.4 Controller Design

Once the description of the system is obtained, a suitable controller must be calculated. In order to do so, a wide variety of techniques are available (e.g. root locus techniques, frequency response techniques, robust control techniques, state-space techniques, etc.) and there is an extensive literature that covers them. What it is relevant for the purpose of this chapter is the fact that the final controller will be expressed as a set of transfer functions or state-space equations, of the form shown in previous section.

### 2.3 Digital Control Fundamentals

As mentioned before, the system to be controlled is normally a physical (continuous) one. The controller acts upon it, normally by means of an electric signal that drives an actuator that will produce the proper action on the system. Initially, analogue devices were used to implement the controller (analogue control system). This approach had two main advantages. First, they work in real time (they are not sampled systems), hence allowing very high bandwidths. Second, neither *analogue-to-digital* (AD) nor *digital-to-analogue* (DA) conversion is needed from or to the controller. Analogue control systems have however some serious drawbacks. Component ageing and sensitivity to environmental changes can severely alter the controller characteristics. Analogue components are also quite susceptible to noise problems and lack programmability. Digital controllers on the other hand offer flexible single-chip embedded solutions based on programmable devices and therefore they can implement algorithmic interlocking and control law execution that allow the controller to adapt itself in real-time to the
system evolution and to the user needs. They are also cheaper and noise immune. In order to integrate a digital controller into a physical system, AD and DA converters must be introduced in the loop. Figure 2.3 shows an example of a digital control system with analogue input and output and the AD and DA converters' placement. Note that the placement of the AD and DA converters may vary from one system to another.

![Figure 2.3: Block diagram of a typical digital feedback control system](image)

It can be seen in Figure 2.3 that the plant is a continuous system while the controller is a digital (sampled) one. Assuming the plant model is defined as $G(s)$, the objective is to calculate a discrete controller that produces the desired closed loop system's dynamics. This can be done, broadly speaking, by three different approaches [12]:

**Emulation.** The controller is designed in the continuous plane ($C(s)$) and then converted (approximated) into $C(z)$ by a number of different techniques (e.g. bilinear transform, pole-zero mapping, etc...).

**Design in the z-plane.** The plant model $G(s)$ is converted (discretised) into $G(z)$ and then the controller is calculated by means of discrete design techniques.

**Design in the w-plane.** It is an hybrid method. The plant is first discretised as $G(z)$ and then transformed back into $G'(s)$ (or $G(w)$), using for example any emulation method. The controller is calculated then as $C(w)$ and then converted by emulation into $C(z)$.

As shall be seen later in this chapter, there are other options to the z-plane, that can be used instead to design a controller by means of the methods described before.
2.3.1 Discrete System Description

Section 2.2.3 presented different ways in which continuous systems can be described. Discrete systems can be represented in a similar way by means of difference equations, discrete transfer functions and discrete state-space equations.

Difference Equations

Problems in which a variable may conveniently be assumed to have only a discrete set of possible values often lead to mathematical models involving difference equations [14]. In sampled systems, such a variable is time. Discrete systems are normally modelled by a set of difference equations. The general form of these equations is:

\[ y[k+n]+a_n-1y[k+n-1]+...+a_0y[k] = b_mu[k+m]+b_{m-1}u[k+m-1]+...+b_0u[k] \] (2.5)

Equivalently to differential equations, if the functions \(a_n-1...a_0\) and \(b_m...b_0\) are constant functions, then Equation 2.5 is defined as a linear difference equation with constant coefficients and represents a LTI discrete system.

Transfer function

In the same way that the Laplace transform can be used to transform differential equations into algebraic ones, other transforms can be used to convert discrete series of values into algebraic equations, thus making more straightforward the study and analysis of difference equations.

The most commonly used is the Z-transform (see Appendix F), closely related with the forward shift operator (see Equation 2.6) by means of its transform variable \(z\).

\[ q(f(t)) \equiv q(f[kT]) \equiv q(f[k]) \equiv qf[k] \equiv f[k+1] \quad \forall T \in \mathbb{Z} \] (2.6)

Probably the most well known transform in the field of digital control and signal processing, it successfully extracts and represents the sequential nature of sampled systems in a very intuitive way. However, this advantage comes at the cost of word length and bandwidth. It is well known that numerical problems arise when using
the shift operator at high sample rates (e.g. [39], [20], [12]). The reason is that the
operator requires to store the actual value of each sample, which forces the coefficients
and state variables to be able to work within a wide dynamic range. As the real-
time requirements force higher sample rates, the dynamic range increases even more.
In order to overcome this problem, control implementations based on the shift operator
are forced in many occasions to use either very large word length or floating point
arithmetic, hence increasing considerably the hardware implementation requirements.

An alternative formulation based on the delta operator (δ-operator) and its related
discrete transform, the Delta-transform (see Appendix F) allows more numerically
robust implementations, especially at high-sample rates, and reduces the hardware
requirements in terms of word length and arithmetic precision. The reason for its
numerically superior properties lies in its definition, based on the differences between
two consecutive sample values, rather than their independent values. There are two
main definitions of the δ-operator in the literature:

- Classic definition [20]:
  \[ \delta(f(t)) \equiv \delta f[k] \equiv f[k+1] - f[k] \]  
  \[ T \]  

- Loughborough definition [15]:
  \[ \delta(f(t)) \equiv \delta f[k] \equiv f[k+1] - f[k] \]  

the definitions being related just by a scaling factor:

\[ \delta_{LBORO} = T \cdot \delta_{CLASSIC} \]  

The first is based upon unifying the continuous and discrete time domains, because
as \( T \to 0 \) with this definition \( \delta \to s \). The second is focussed upon the practicalities of
implementation for which the division by \( T \) will always be combined with the coeffi-
cients, hence the simpler formulation. Since this research is concerned with implement-
tation, the Loughborough definition of the δ-operator is used throughout this thesis
and from now on it will be referred simply as δ-operator. This operator is related with
the forward shift operator as follows:

\[ \delta = q - 1 \]  

\[ (2.10) \]
The \( q \)-based transfer function of the LTI system defined in Equation 2.5 (for zero initial conditions) is:

\[
Y(z) = \frac{b_m z^m + b_{m-1} z^{(m-1)} + \ldots + b_1 z + b_0}{z^n + a_{n-1} z^{(n-1)} + \ldots + a_1 z + a_0}
\]  

(2.11)

The \( \delta \)-operator-based transfer function of the same system can be expressed as:

\[
Y(\gamma) = \frac{b'_m \gamma^m + b'_{m-1} \gamma^{(m-1)} + \ldots + b'_1 \gamma + b'_0}{a'_n \gamma^n + a'_{n-1} \gamma^{(n-1)} + \ldots + a'_1 \gamma + a'_0}
\]  

(2.12)

where \( \gamma \) represents the Delta-transform variable. Appendix F provides a detailed description of the relation between the coefficients in Equation 2.11 and Equation 2.12.

**State-Space Models**

The discrete state space representation of a system follows the same structure presented in Equations 2.3 and 2.4. In this case however, the different states are related by a series of first order difference equations rather than first order differential equations. The general structure of a LTI discrete system with \( n \) state variables, \( r \) inputs and \( p \) outputs can be therefore expressed as follows:

\[
x[k+1] = F \cdot x[k] + G \cdot u[k]
\]  

(2.13)

\[
y[k] = C \cdot x[k] + D \cdot u[k]
\]  

(2.14)

where \( x = n \)-vector, \( y = p \)-vector, \( u = r \)-vector, \( A = (n \times n) \) matrix, \( B = (n \times r) \) matrix, \( C = (p \times n) \) matrix, \( D = (p \times r) \) matrix.

Similar to the continuous case, there is an infinite number of state-space representations of the same system, but again a specific set of these possible representations is particularly useful in control theory.

In this thesis, the modified delta canonic form presented in Equation 2.15 and Equation 2.16 will be used due to its numerical benefits particularly at high sample rates. Chapter 4, will describe this canonic form in detail.
2.3.2 Adaptive Control

An adaptive control system uses a control scheme that is capable of modifying its behaviour in response to changes in the process dynamics [30]. Adaptive controllers have been extensively used in several industries including chemical, aerospace, automotive, and pulp and paper. There are three well known adaptive control schemes: gain scheduling, model-referenced adaptive control (MRAC) and self-tuning regulators (STR). All these schemes adapt the behaviour of the controller by providing it with new sets of coefficients to improve system response at this particular moment, but differ in the way these coefficients are calculated. The adaptation algorithm runs in parallel with the control law execution, usually at much lower frequency. Adaptive control is widely covered in the literature and it is out of the scope of this thesis to describe further this subject. However, gain scheduling is a fairly simple adaptive technique yet suitable in many applications and will be briefly described below.

Gain scheduling

This type of adaptive control system is based on the adjustment of the controller parameters in response to the operating conditions of the process [5]. This control scheme is particularly useful when the variations in the process dynamics are predictable. Gain
scheduling can be regarded as a mapping from the process parameters to the controller parameters [30]. In practice, a gain scheduler can be implemented as a look-up table.

![Block diagram of a gain scheduling controller](image)

Figure 2.4: Block diagram of a gain scheduling controller

Examples of the application of this control scheme can be found in most of the fields. One example is suspension control in railway systems, where although the nominal mass of the vehicle remains constant, the number of passengers produces a variation in the total mass of the system that cannot be neglected. In this case, gain scheduling can be applied by calculating a number of sets of control coefficients for a range of mass variations (number of passengers). Another example can be found in aerospace applications where the operating conditions of an aircraft vary for example with the height (e.g. oxygen amount, air density, etc.). Again, a number of sets of coefficients can be calculated off-line and applied depending on the actual flight conditions (i.e. height in this particular example).

2.3.3 Intelligent Control

Fuzzy logic, neural networks, and evolutionary computing have provided important tools and techniques for system control. Intelligent control seeks to achieve good performance in machines, industrial processes, consumer products and other systems by using control approaches that, in a loose sense, tend to mimic direct control by experienced humans. Many of these techniques can learn and compensate for parameter changes and disturbances, and are able to provide satisfactory control even in incompletely-known and unfamiliar situations [30]. Although interesting, these tech-
niques are especially useful in highly non-linear systems and/or very ill-defined models. This thesis focuses on well-defined real-time embedded LTI and LTV systems, and therefore intelligent control is out of its scope.

2.4 Digital Filtering and Signal Processing

Current control systems require more than simple control law executions. Digital filtering and signal processing are common parts of modern control implementations. Examples can be found in a variety of fields such as aeronautics and mechatronics. The mathematical representation of digital filters and different signal processing techniques share many common aspects with digital control, specifically their matrix-based formulation and the intensive use of multiply-accumulate (MAC) operations.

The remaining of this section briefly describes some of the key digital signal processing techniques in order to better understand their similarities with the digital control structures, in terms of numerical requirements and mathematical formulation.

**Digital filtering.** Digital filters are broadly divided into two classes, namely infinite impulse response (IIR) and finite impulse response (FIR) filters. Either type of filter can be represented by its impulse response sequence, \( h(k) (k = 0, 1, \ldots) \). In both cases, the input and output signals to the filter are related by the convolution sum, which is given in Equation 2.17 for an IIR and in Equation 2.18 for an FIR filter [26].

\[
y(n) = \sum_{k=0}^{\infty} h(k)x(n - k)
\]  
\[
y(n) = \sum_{k=0}^{N-1} h(k)x(n - k)
\]

Due to the infinite length of the computations required by Equation 2.17, IIR filters are expressed in a recursive form (hence they are often called recursive filters):

\[
y(n) = \sum_{k=0}^{\infty} h(k)x(n - k) = \sum_{k=0}^{N} b_k \cdot x(n - k) - \sum_{k=1}^{M} a_k \cdot y(n - k)
\]  

As can be noted, the output sample, \( y(k) \), of an IIR filter is a function of past outputs as well as present and past input samples, while in a FIR filter its output only depends
on present and past inputs. This dependence (or not) on past output values, makes it possible to assure the stability of a FIR filter, a certainty that cannot be achieved in a IIR filter. Figure 2.5 shows a block diagram representation of a second order IIR filter in direct realisation. As can be seen, digital filters can be represented in the same way as discrete SISO control systems and they are often formulated as transfer functions with the structure of Equation 2.11. A general FIR filter can be represented using the same structure shown in Figure 2.5, just by defining the output coefficients as follows: $a_0 = 1$ and $a_i = 0$ for $i = 1, 2, ...$

**Convolution.** Convolution is one of the most frequently used operations in digital signal processing, being for example the basic operation in digital filtering. An important characteristic is the fact that convolution in the time domain is equivalent to multiplication in the frequency domain.

Given two finite length sequences, $x(k)$ and $h(k)$, of length $N_1$ and $N_2$, respectively, their linear convolution is defined as:

$$y(n) = h(n) \otimes x(n) = \sum_{k=-\infty}^{\infty} h(k) \cdot x(n - k) = \sum_{k=-0}^{M-1} h(k) \cdot x(n - k) \quad (2.20)$$

where $n = 0, 1, ..., M - 1$ and $M = N_1 + N_2 - 1$

**Correlation.** It is frequently necessary to be able to quantify the degree of interdependence of one process upon another, or establish the similarity between one set of data and another. In other words, the correlation between the processes or data is sought [26]. Applications are found in robotic vision and remote sensing by satellite amongst others. The correlation can be calculated in a number of ways, depending
Chapter 2: Background on Digital Control I. Controller Formulation

on the nature and characteristics of the processes to be correlated. As an example, Equation 2.21 describes the correlation between two waveforms of the same length.

\[ r_{12}(j) = \frac{1}{N} \sum_{n=0}^{N-1} x_1(n) \cdot x_2(n + j) \]  

(2.21)

**Discrete transforms for Signal Processing.** Discrete transforms allow the representation of discrete-time signals in the frequency domain or the conversion between time and frequency domain representations. They are extensively used in signal and multimedia processing applications.

### 2.5 Summary

Continuous and digital systems can be described mathematically in a number of ways. This chapter briefly presented the most common of these formulations.

Although most designs are based on the shift operator \((q)\), its numerical issues, especially when sampling at much faster rates than the system bandwidth, make it a less-than-recommended option for use in high-performance real-time systems. Instead, the delta operator \((\delta)\) is recommended for a number of reasons, namely it is numerically more robust at high-sample rates, allows better hardware implementations and, as will be shown in Chapter 4, in its modified canonic form, the resultant coefficients are much better scaled, reducing the dynamic range required to represent them.

Contemporary control systems require much more than just control law execution. Often, signals are digitally processed and based on them certain inputs are provided to the controller. As presented in this chapter, the arithmetic structure of signal processing algorithms doesn't differ much from the required one to process control laws and it is fundamentally based on the dot product between a matrix and a vector. MAC operations are extensively used in all digital signal processing applications (including digital control) since they best match this matrix-based processing. Therefore digital signal processing algorithms are best mapped into hardware architectures that efficiently process such operations.
Chapter 3

Background on Digital Control II.
Processor Architectures

3.1 Introduction and Objectives

Once the plant is mathematically described and the appropriate digital algorithm is developed and simulated, the next step leads to the physical implementation of such algorithm, usually as a set of instructions running on some sort of processing architecture. As mentioned before, digital control algorithms can be regarded as a subset of the signal processing algorithms and therefore they are typically implemented into digital signal processing hardware architectures. This architectures should be able, at least, to receive sampled data, process it and generate the proper output to be sent back to the system. In the case of embedded real-time control, the architecture usually needs to be fast enough to achieve high-sample rates as well as size and power consumption aware.

This chapter reviews different issues related to the actual hardware implementation of signal processing algorithms. The objectives are to:

- Understand the processing requirements that are particular to signal processing algorithms, paying special attention to the embedded real-time control scenario.
- Gain a background knowledge on computer architecture for signal processing in
order to better understand the different design options, that can best match the processing requirements previously presented. This review of computer architecture for signal processing will be useful to understand the design decisions that lead to the embedded control solution presented in Chapter 5.

- Review major embedded architectures and devices used for implementing digital processing algorithms, identifying their advantages and drawbacks especially when operated at high sample rates under embedded real-time requirements.

3.2 Implementation Requirements for Embedded Real-Time Digital Controllers

Digital signal processing algorithms in general and digital control algorithms in particular make an extensive use of arithmetic operations with intensive data flow through the CPU. The efficient execution of their algorithms in real-time requires a hardware architecture and instruction set radically different from those of standard microprocessors [26]. The embedded space has its own particular requirements in terms of power consumption and gate count. This section presents a summary of the main special characteristics common to the majority of these algorithms that the hardware architecture should efficiently address in order to achieve good performance while meeting the embedded space constraints.

3.2.1 Input/Output Requirements

Usually, embedded controllers are required to interface with both analogue and digital devices since they can be placed between digital stages or receive/send signals from/to different types of analogue sensors/actuators. In order to interface with analogue systems, a digital system requires two conversion processes: analogue-to-digital conversion (ADC) at the inputs, and digital-to-analogue conversion (DAC) at the outputs.

In embedded systems, size is always a constraint and analogue circuitry is especially
“voluminous” when compared with digital gates. Thus, a controller architecture that includes ADC/DAC logic in its core would reduce the external components of the final system and therefore would be more suitable for embedded applications. However, a fixed number of converter units could lead to a waste of resources since different systems have different input/output requirements. Hence, the possibility of configuring the number of digital inputs and outputs of the controller, as well as its number of converter units, would offer a tailored solution with fit-to-purpose logic and resources. This solution would be smaller in size and would consume less power.

System-on-chip solutions implement different functional modules on a single chip. Some of these modules need to interact between each other at a high-level. For example, an embedded CPU may be used for coordinating some other (more targeted) modules. Therefore, in order to achieve a seamless integration on a SoC solution, a controller should implement some industry standard bus interface.

3.2.2 Arithmetic Requirements

Digital processing architectures generally deal with infinite continuous streams of data [22] that must be processed in real-time. Arithmetic operations are intensively used while branches, interrupts and other instructions common to desktop and server systems are reduced to a minimum. In fact, as can be seen in the particular case of control law execution and digital filtering, many arithmetic operations are performed before any new sample is considered or any new action calculated. Thus, the main processing load in a digital controller falls upon its AU, which must be carefully optimised since it will probably dictate the system’s performance. Next, the key arithmetic characteristics required in most embedded digital signal processing architectures are reviewed:

Fast Multiply Accumulation (MAC) Operation Execution

The basic numerical operations in digital signal processing are multiplications and additions. Multiplication, in software, is notoriously time consuming, while additions are even more time consuming if floating point arithmetic is used [26]. Due to the
extensive use of matrices in digital signal processing, the most common arithmetic operation to be performed is of the type:

$$y = \sum_{i=1}^{N} c_i \cdot x_i$$  

(3.1)

which represents for example the dot product between a vector of coefficients \([C]\) and the current vector of state variables \([X]_k\) in a typical discrete state space controller representation (see for example Equation (2.14)).

A MAC operation returns the result of multiplying two numbers and adding the result to a third one as shown in Equation (3.2).

$$d = a \cdot b + c$$

(3.2)

Equation (3.1) is best mapped by means of MAC operations. Therefore, in order to achieve real-time performance in digital signal processing, dedicated hardware MAC units must be implemented. The effectiveness in the execution of MAC operations is the one of the primary peak performance metric used by most DSP architects and is usually measured in MACs/second.

Arithmetic Format

One of the most fundamental characteristics of a programmable digital signal processor is the type of native arithmetic it uses. Many DSPs use fixed-point arithmetic [26], where numbers are represented as integers or as fractions in a fixed range, while others use floating-point arithmetic. In this case, values are represented by a mantissa and an exponent as shown in Equation (3.3):

$$value = mantissa \cdot 2^{(exponent)}$$

(3.3)

The mantissa is generally a fraction in the range [-1.0 to +1.0], while the exponent is an integer that represents the number of places that the binary point (analogous to the decimal point in a base 10 number) must be shifted left or right in order to obtain the desired value [6].

Fixed-point arithmetic is the most prevalent in digital signal processing applications because it leads to fast and inexpensive implementations, but it is limited in the range
of numbers that can be represented and in its accuracy, and is also susceptible to problems of overflow [26].

Floating-point arithmetic is preferred when the magnitudes of varying variables or system coefficients vary widely (wide dynamic range) and/or when high precision is required. These advantages simplifies programming because algorithms developed and simulated in larger machines (e.g. desktop computers) in a high-level language, can be directly implemented in an embedded processor that supports such arithmetic with little changes to the core of the algorithm. However, floating-point arithmetic is more expensive and slower that its fixed-point counterpart. Although one of the most widely used binary floating-points systems is the IEEE 754 standard [25], sometimes the dynamic range and accuracy provided by its 32-bit single precision representation is not required and a more compact and faster AU could be designed based on a reduced word length floating-point arithmetic. However, a reduction in the word length used to represent the system parameters (or controller coefficients) can increase the quantisation errors, and therefore this approach should be carefully assessed.

Quantisation is the process of mapping real numbers into a finite word length representation. This process introduces inevitably an error common to most digital signal processing implementations (quantisation error) that cannot be removed [26]. In control systems design the coefficients are usually calculated with 32-bit floating-point precision by a software tool running on a desktop computer, but afterwards they often need to be quantised to fit into the word length of the processing architecture selected for implementation. Another source of quantisation error in control algorithms is the ADC process, where an analogue signal must be represented by a finite number of bits.

The precision of both fixed-point and floating-point arithmetic representations depends on the number of bits used for representing them. Choosing a technology with the right word size is an important implementation decision that should be carefully considered. A practical method for determining the coefficient word length in digital filters can be found in [16].
Internal variables overflow

For a successful implementation, it must be noticed the fact that internal variables within recursive filters can increase to many times the size of the input, even though the final output may be similar in size to the input [12]. This can be applied to many other algorithms, where an intermediate operation can surpass, by several times, the size of the input. In a two's complement fixed-point format, the addition of two large numbers of a similar sign may produce an overflow, that is a result that exceeds the permissible word length, which would cause a change in its sign. Thus, a very large positive number becomes a very large negative one and vice versa (Figure 3.2(a)). In this case restoration of normal operation is by no means assured. This situation can be avoided in most cases by extending the variables word length with enough extra guard bits. Figure 3.1 shows the example of a 6-bit variable encoded in 2's complement with three extra guard bits. Notice that the sign bit must be shifted at the most significant bit position.

![Guard Bit Diagram](image)

Figure 3.1: Overflow prevention example by adding three extra guard bits

This technique allows the architecture to implement saturation arithmetic by checking the sign of the guard bits and the most significant bit of the original 6-bit variable, while preserving at the same time the accuracy of the calculations for a range $2^n$ times bigger than the one allowed without extra bits, where $n$ is the number of guard bits.

Output Saturation

Digital signal processing algorithms often rely on saturating arithmetic and would be incorrect if run on a device without it [22]. Even with a well chosen coefficients' and internal variables' word length, an output overflow can (although occasionally) still happen and therefore it is common practice to use saturation arithmetic. This
technique simulates the saturation effects typical to analogue devices such operational amplifiers by setting the outputs to their maximum permissible positive or negative value when an overflow is detected. Figure 3.2 shows how two systems output the value stored in a 6-bit variable implementing three guard bits as shown in Figure 3.1. The output is 6-bit wide expressed in 2's complement. The first system (a) doesn't implement saturated output logic, while the second one (b) does. In these figures, $x$ represents the correct output (as stored in the extended version of the original 6-bit variable), while $y$ represents the actual output.

![Figure 3.2](image)

Figure 3.2: Comparison between the output of two systems. The first one (a) doesn't implement saturated output logic, while the second one (b) does.

### 3.2.3 Other Requirements

A requirement common to most commercial products that are to be produced in volume is the price per unit. In the particular case of digital control, designers try to use the lowest cost DSP architecture that can provide acceptable performance for a particular application, even though such devices may be considerably less flexible, more difficult to program, and less performing than other more adequate (and expensive) ones. Among DSPs and MCUs from the same family, the least expensive family members tend to have significantly fewer features, less on-chip memory, and lower performance than the more expensive members [6].

Sometimes, new features need to be implemented on an existing product that requires higher computational power. This is usually addressed by choosing a faster
processing device. However, is a common requirement to reuse the existing Printed Circuit Board (PCB) layout which forces the new device to be pin-compatible with the previous one (i.e. they will probably be from the same manufacturer and family).

Other requirements commonly found in embedded systems are design size and power consumption.

3.2.4 FPGA-based designs

Although embedded digital controllers have been traditionally implemented using off-the-shelf devices, FPGA technology can certainly change the way digital control is understood. Reusability has not been common in the embedded applications, due to the targetted nature of their design and the constraints in space and resources. Often, changes in a plant requires the design of a new embedded controller with all the associated external circuitry.

FPGAs introduce a new dimension to the concept of embedded hardware design: reconfigurability. A design based on FPGA technology, would allow addressing plant changes at gate level, as many times as needed, testing the new embedded architecture directly on the plant. This could also allow for periodic and non-intrusive updates and upgrades of the controller. In order to achieve this goal, the control architecture should be designed as a soft intellectual property (soft IP core), synthesisable for a variety of FPGA technologies (in order to avoid vendor-specific restrictions).

3.3 Basic Concepts on Computer Architecture for Digital Signal Processing

This section briefly covers the main concepts of computer architecture, putting special attention on the particularities that can be found on devices targeted to digital signal processing. It introduces basic concepts that will later be required in order to understand the architecture and implementation decisions taken in the design of the rCSP. This section is based on [22] which is probably one of the most acknowledged
publications in computer architecture and it is used as reference in many other books and related works in the field. Therefore, it is strongly recommended as reference in case additional information were required.

3.3.1 Instruction Set Architecture (ISA)

The ISA of a computer system is the portion of the computer visible to the programmer or compiler writer [22]. A typical classification of the main ISA characteristics is summarised below.

**Internal storage type**

The type of internal storage in a processor is perhaps the most basic differentiation between architectures [22]. The following internal storage classification refers to the way the data is brought into the *arithmetic logic unit* (ALU) and sent back to the memory: a) Stack, b) Accumulator, c) Register-memory, d) Register-register/load-store e) Memory-Memory.

The load-store architecture doesn't allow memory references in ALU instructions. Thus, data is retrieved from the memory and stored in a general purpose register (by means of a *LOAD* instruction) before it enters the ALU. In the same way, after executed, the result of the operation is stored in a general purpose register and it is sent back into the memory by means of a *STORE* instruction. Most recent RISC architectures (e.g. ARM, MIPS, PowerPC, SPARC and TRIMEDIA TM5200, etc.) implement this internal storage type.

A number of advantages can be found in this approach. First, registers are internal storage element in the processor, and therefore accessing them is much faster than accessing an external memory. Second, registers are more efficiently managed by a compiler than other forms of storage. In general, when registers hold variables the overall memory traffic is reduced, the programme execution speeds up and the code density improves.
Memory Addressing

Independently of the storage architecture, the different ways an instruction can refer to one or several memory addresses must be defined, i.e. how many memory addresses are interpreted and how they are specified. Addressing modes have the ability to significantly reduce instruction counts; on the other hand they also add complexity to the processor design and may increase the clock cycles per instruction (CPI). Table 3.3.1 summarises the addressing modes that are most frequently used by a wide variety of applications typically run on general purpose processors.

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R5</td>
<td>Reg[R4] ← Reg[R4]+Reg[R3]</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>Reg[R4] ← Reg[R4]+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>Reg[R4] ← Reg[R4]+Mem[100+Reg[R1]]</td>
</tr>
</tbody>
</table>

Table 3.1: Most commonly used addressing modes in general purpose processors

The structure of digital signal processing algorithms deviates from the one found in standard programs run on a general purpose processor. In order to optimise the execution of such algorithms, architectures targeted towards digital signal processing implement new addressing modes specifically designed to reduce the address calculation overheads or the memory accesses, therefore improving performance. Some examples of this targeted addressing modes are:

- **Modulo or circular** addressing, optimised for handling the data address calculation in circular buffers.

- **Autoincrement and autodecrement addressing.** These addressing modes are usually implemented in combination with a variety of other ones such as circular addressing, with preincrement, with postincrement, etc.

- **Bit reverse** addressing, specifically design for the data arrangement that happens at the beginning or end of the Fast Fourier Transform (FFT) processing algorithms.
Regardless of the clear advantages that these addressing modes provide to the execution of signal processing algorithms, probably due to the increasing size of their applications, engineers and programmers more and more often rely on high-level (C/C++) compilers. These compilers are very limited in their capabilities to spot sections of C/C++ code that can be optimised by the use of these novel addressing modes. In fact most often the only way of making use of them is by coding a particular routine directly in assembler.

An example that summarises this mismatch between the hardware and compilers can be found in a typical DSP architecture, namely Texas Instrument's TI TMS320C54x. After executing a benchmark of 54 DSP routines, the following addressing result were obtained: From the 17 different addressing modes (not counting register access) available in this architecture, only 6 of them account for 95% of the addressing. Interestingly, these 6 addressing modes can be found in desktop and server architectures, and they are not targeted to signal processing. In fact, four addressing modes found in MIPS architecture account for 70% of the modes. For a detailed description of this test and benchmark see [22] and [27].

Type and Size of Operands

Signal processing algorithms are characterised by a predominant use of arithmetic operations and therefore they require a more reduced set of operand types than general-purpose architectures. Their sizes tend to be smaller and more flexible, being sometimes not a power of 2 (e.g. 24-bit architectures). Often, 16-bits operands are more used even in 32-bit architectures. Reduced operand size usually implies an increment in code density, something very much looked for in embedded applications.

Operations for Media and Signal Processing

Table 3.2 categorises the different operators supported by most instruction set architectures.

In many signal and media processing applications, the accuracy is measured by
human perception (e.g. image and sound quality). In digital control and filtering the accuracy depends on many factors (e.g. plant model deficiencies, noise, non-linearities, number and word length of the coefficients used in the filter, etc.). Furthermore, the sensors' resolution and actuators' accuracy can be seen as disturbances introduced in the system. These particular relaxations in the accuracy requirements can be exploited to obtain higher performing architectures. For example, signal processing applications often run on processors with 32-bit (or even 64-bit) data word, but their precision only requires operating on 16-bit or 8-bit integers. In order to avoid wasting resources and speeding up the algorithms' execution, these processors tend to implement a number of partitioned ALUs (i.e. ALUs that can operate on several narrower data items at the same time). In that way, a 32-bit partitioned ALU can execute for example four 8-bit adds on a single clock cycle. These operations are called Single Instruction Multiple Data (SIMD) or vector instructions and they will be further explained in Section 3.4. Another characteristic particular to these applications (especially in the case of digital control) is the use of saturated arithmetic and the possibility of rounding the values before being outputted. Due to the intensive use of MAC operations, the target kernels of most specialised processors (e.g. DSPs, MCUs) implement a MAC instruction. Often, such instructions are executed in specialised hardware (MAC units) with a single-cycle execution.

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic and logical operations</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Floating-point operations</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Load-Stores</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jumps, procedure call and return, traps</td>
</tr>
<tr>
<td>System</td>
<td>O.S. calls, virtual memory management instructions</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal operations</td>
</tr>
<tr>
<td>String</td>
<td>String move, compare, search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Pixel and vertex operations</td>
</tr>
</tbody>
</table>

Table 3.2: Categories of instruction operators
Instructions for Control Flow

Four types of control flow changes can be distinguished, namely conditional branches, jumps, procedure calls and procedure returns. Changing the control flow implies certain overheads that will depend on the architecture. Minimising these overheads and enforcing execution predictability is a crucial task in real-time applications. Most of the changes in control flow are due to conditional branches, and in many cases the test condition is a simple comparison (very often with zero). In order to exploit this fact, many architectures implement compare and branch instructions. Another control flow structure targeted for the particular characteristics of signal processing algorithms is a looping structure often represented by repeat instruction. This assembly instruction allows a single instruction or a block of them to be repeated a number of times. Some architectures allow them to be nested up to a particular level. A single instruction repetition example for Motorola's DSP560000 processor [26] is:

```
REP #N-1
MAC X0,Y0,A
```

and a block repetition example for the Texas Instruments' TMS320 family is:

```
RPTB loop
.
.
loop (last instruction)
```

These instructions not only can provide zero-overhead looping but also produce a more compact code, both improvements being greatly appreciated when implementing embedded real-time applications.

Encoding an Instruction Set

Instructions are encoded into a binary representation suitable for their execution in the processor. This representation affects not only the size of the compiled program,
but also the implementation of the processor, which must decode this representation to quickly find the operation and its operators. The encoding will also define the different range of addressing modes that can be supported and the degree of independence between opcodes and addressing modes.

There are three main types of instruction encoding:

1. *Variable.* This format can support any number of operands. It generally enables the smallest code representation, since unused fields are not included. As a drawback, the instruction decoding can become quite complex and make a pipelined implementation more difficult. This encoding is used for example in the *Intel 80x86* architecture.

2. *Fixed.* This format always has the same number of operands. It allows a faster and easier decoding, and it is used in many RISC architectures like ARM, MIPS, PowerPC and SPARC. As a downside, fixed-format representations can increase the application code size.

3. *Hybrid.* This approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address. This format is the choice of many architectures targeted to embedded applications for it provides a significant reduction in code size. Examples are Thumb, MIPS16 and Texas Instruments' TMS320C54x architectures.

### 3.3.2 Memory Hierarchy

The great performance gap between CPUs and the memories has been exponentially growing for the last twenty years. In addition to that, the ever-increasing memory space requirements of most software applications (whether embedded, desktop, server, etc. ones) led to the design of a hierarchical memory structure. One of the key elements in this hierarchy is the cache memory. Instruction caches are widely implemented in embedded processing architectures used in real-time applications. The reason is the predictability in the code behaviour typical to most signal processing algorithms. Cache memory not only improves the performance of the processor (in terms of latency)
by reducing the number of accesses to the main memory, but also decreases the power consumption [22]. Accessing on-chip memory is much more efficient than it is to drive the pins of the chip and the memory buses, activate the external memory chip's components and bring the data back to the chip.

Most general purpose CPUs are based on a pseudo Harvard architecture, which is a term applied to machines with a single main memory but with separate instruction and data cache. DSP devices on the other hand, usually implement what is called a modified Harvard architecture. This architecture implements separated data and address bus and distributes the data memory in separate blocks of similar algorithmic characteristics. This is a very suitable structure for control and signal processing applications, allowing simultaneous access to different memory elements (e.g. coefficients and variables) which is particularly useful in arithmetic intensive algorithms.

3.3.3 Parallelism

A single device can achieve a certain level of parallelism by overlapping the execution of two or more instructions, by performing several arithmetic operations simultaneously or by issuing a number of instructions on the same clock cycle. These approaches can be combined in different ways, obtaining, in the ideal, case a linear increase in performance. Sections 3.4 and 3.5 will review the main techniques used to achieve parallelism on a single device.

3.4 Operation-Level Parallelism.

Single Instruction Multiple Data (SIMD)

SIMD processing is used to increase the number of operations performed per instruction [26]. Typically, there are two ways (often combined) of implementing SIMD processing:

Multiple execution units. This approach is based on the replication of certain execution units (e.g. ALUs, MACs, Shifters, etc.) to process blocks of data si-
multaneously, increasing in that way the number of operations performed per clock cycle. Multiple data paths are required in order to drive the data from the memory to the different execution units.

**Partitioned ALU.** Another way to achieve operation-level parallelism is by partitioning the execution units and performing on them multiple operations on smaller data sized operands. Figure 3.3 presents an example of partitioned MAC unit. In the first case (a) the unit works in full size (one operation is executed), while (b) shows the same unit working partitioned (up to four operations can be executed simultaneously). Note that only when the ALU works in full size, all its logic is used.

![Diagram of partitioned ALU](image)

Figure 3.3: An example of SIMD processing by partitioning the execution units. In this example, the execution unit in (a) can be partitioned as in (b) so up to four 8-bit operations can be executed simultaneously on two 32-bit word inputs.

SIMD instructions are actually an abbreviated version of an architecture style that has its own compiler technology, namely vector computers. Programmers implementing algorithms on SIMD architectures face a lack of compiler support for SIMD instructions [22]. Normally, in order to take advantage of operation-level parallelism, the specific set of SIMD assembly instructions must be explicitly used, since generally C/C++ compilers are not capable of vectorising the code automatically. SIMD processing clearly enhances vector operation performance, but its coding is still not straightforward.
Chapter 3: Background on Digital Control II. Controller Implementation

3.5 Instruction-Level Parallelism (ILP)

The potential overlap of instructions that certain design techniques provide is called instruction-level parallelism (ILP), since the instructions can be evaluated in parallel [22]. These techniques will be now briefly described, and classified following a natural division between processors, based on the number of instructions that they are able to issue during the same clock cycle. In order to understand how ILP can be exploited, it is important to understand the following division: The different dependencies that exist on a segment of code are exclusive properties of the programme that is being executed. On the other hand, how these dependencies are handled (i.e. whether they are detected as hazards and whether those hazards actually cause a stall) are strictly properties of the pipeline organisation.

3.5.1 Single-Issue Architectures

Pipelining

All processors since about 1985, including those for embedded applications, use pipelining to overlap the execution of instructions and improve performance. Pipelining takes advantage of the parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs [22]. In a perfect pipeline, the average time per instruction (ATPI) is given by Equation 3.4

\[ ATPI = \frac{\text{time per instruction (nonpipeline)}}{\text{number of pipeline stages}} \]  

(3.4)

In the ideal case, the speed increase is equal to the number of pipeline stages. In practice, the speed increase will be less because of the overheads in setting up the pipeline, the delays in the registers, the non-symmetry of the design, and so on.

The ideal speedup gained from pipelining can be calculated as:

\[ \text{speedup} = \frac{\text{average instruction time (nonpipelined)}}{\text{average instruction time (pipelined)}} \]  

(3.5)

However, the pipelining of a processor doesn’t come at no cost. There are situations called hazards, that prevent the next instruction in the instruction stream from
executing during its designated clock cycle. Hazards reduce performance from the ideal speedup gained by pipelining. There are three classes of hazards [22]:

1. **Structural Hazards** that arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. **Data hazards** that arise mainly when an instruction depends on the result of a previous instruction that is still being processed at certain stage inside the pipeline. In complex pipeline architectures, *name dependencies* can also produce data hazards.

3. **Control hazards** that arise from the pipelining of branches and other instructions that change the *programme counter* (PC) of a processor.

Normally pipelined processors implement certain hardware structures to prevent the pipeline from stalling, but they don’t always succeed. Stalls increase the average number of clock cycles required to execute an instruction, thus reducing the processor’s performance. Careful programming and certain compiler techniques allow the effect of these hazards to be reduced.

### 3.5.2 Multi-Issue Architectures

The goal of multi-issue processors is to allow multiple instructions to be issued in a clock cycle [22]. These processors can be divided into two basic types: *superscalar* processors and *VLIW* (very-long-instruction word) processors. The order in which the instructions are issued on these processors provides the following classification:

**Statically Scheduled Processor.** Also called “in-order execution”. The order in which the instructions are executed is fixed. It relies on compiler techniques, for it is the compiler that decides the execution order of the instructions. On some architectures (superscalar) the hardware decides how many instructions can be issued simultaneously (*dynamic issue capability*). For other architectures
(VLIW), the compiler is in charge of deciding how many instructions are issued at a time (static issue capability).

**Dynamically Scheduled Processor.** Also called “out-of-order execution”. In this scheduling method, the hardware decides “on the fly” which instructions are issued on every clock cycle. It has the traditional benefit of boosting performance in case of data hazards, but it also allows the processor to potentially overcome the issue restrictions [22]. An extension of this mechanism is the *speculative dynamic scheduling*, where the selected instructions are fetched, issued and executed as if the branch prediction module were always correct. It combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependencies are resolved, and dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

As mentioned above, VLIW and superscalar implementations are the two major multi-issue approaches used by processor designers. Their main difference lies on the level of support they require from the compiler. In VLIW architectures, the performance is tightly related to the the compiler’s efficiency in optimising parallelism. Superscalar architectures in contrast, implement hardware modules in charge of discovering such parallelism among instructions at issue time. Therefore, superscalar architectures inherit a certain degree of unpredictability in their instructions’ execution order which is usually unsuitable for real-time applications. Following, both implementations are briefly reviewed.

**VLIW**

Each very-long instruction (sometimes called super-instruction) is essentially a concentration of several shorter instructions. VLIWs use multiple, independent functional units running in parallel to carry out all the instructions packed on the super-instruction on a single cycle. Generally, most VLIW architectures support SIMD operations. This means that in the same way that a VLIW instruction can execute several instructions in parallel, each of these parallel instructions can execute a number
of arithmetic operations simultaneously. Figure 3.4 depicts the simultaneous issue of four instructions, each one being capable of executing $n$ arithmetic operations.

![Diagram of VLIW instruction execution](image)

Figure 3.4: A typical structure and execution of a four-issue VLIW instruction. AO stands for arithmetic operation.

In an ideal case, an $M$-issue VLIW architecture with a maximum of $n$ arithmetic operation per instruction is capable of executing up to $M \cdot n$ arithmetic operations simultaneously.

VLIW processors rely entirely on the compiler to achieve good execution performance. They have static-issue capabilities because the compiler is in charge of deciding how many instructions are issued at a time (from 0 to $M$ on a $M$-issue processor). They are statically scheduled architectures since the order of execution is determined at compilation time (i.e. first Instruction 1, then 2, then 3, and then 4 in the example shown in Figure 3.4). The main advantages that this compiler based execution offers are in terms of hardware reduction and simplicity.

VLIW implementations are simpler than superscalar ones. Just as RISC architectures permit simpler, cheaper high-performance implementations than CISC architectures, some VLIW architectures can be simpler and cheaper than RISC architectures because of hardware simplifications [47].

Some examples of VLIW architectures targeted for signal processing are Texas Instruments' TMS320C62x family, Adelante's Saturn DSP Core and Philips' TriMedia.
Superscalar

Superscalar processing is another technique for increasing the instruction rate of a processor by exploiting instruction level parallelism. They all implement a hardware module in charge of deciding how many instructions can be executed for every particular clock cycle. As mentioned at the beginning of this section, these processors can be classified as statically scheduled (if the order of execution is fixed at compilation time), or dynamically scheduled (if the execution order is decided “on the fly” by the hardware). In superscalar architectures, multiple execution units are provided and several instructions may be issued to the units for concurrent execution [26].

Examples of such architectures can be found in general purpose processors like Pentium III/4, MIPS R10K, Alpha 21264, and Analog Devices’ TigerSHARC DSP.

3.6 Devices and Architectures for Digital Control

The implementation of digital signal processing algorithms in real-time requires both hardware and software elements [26]. The software may be written in assembly language (frequently used in the time-critical segments of the code in order to speed up their execution), or high-level language (usually C or C++). The hardware may be a single (or an array of) standard microprocessor, DSP chip, microcontroller or microprogrammed special purposed architectures. These devices have the following characteristics:

3.6.1 General Purpose Embedded Micro Processor

These devices are design to support the execution of a wide variety of applications, ranging from word processors running on a PDA to 3D real-time games running on a game console.

Nowadays many high-end CPUs have been enhanced to increase the speed of computations associated with signal processing tasks. Perhaps the most common modification is the addition of SIMD-based instruction set extensions, such as MMX, SSE
Chapter 3: Background on Digital Control II. Controller Implementation

(Streaming SIMD Extension) and SSE2 for the Pentium, and AltiVec for the PowerPC [11].

Using this approach, general-purpose processors are often able to achieve performances on digital signal processing algorithms that are even better than the fastest DSP processors. This surprising result is partly due to the effectiveness of SIMD operations, but also because many CPUs operate at extremely high clock speeds. Nevertheless, their power consumption, heat dissipation issues, size and external logic requirements make them unfit for embedded applications. Furthermore, the superscalar architectures and dynamic features common among high-performance CPUs can be highly unsuitable for certain real-time applications.

3.6.2 Digital Signal Processor (DSP)

DSPs are microprocessors whose design is optimised for digital signal processing - the mathematical manipulation of digitally represented signals [6]. Architecturally they are often quite different from normal microprocessors. Some key features of the particularities of their design are:

- **Harvard Architecture.** Separate data and address busses are provided to improve memory bandwidth by allowing a new instruction to be fetched while simultaneously executing the current one. Often, two or more separate banks of data memory are implemented, each of which is accessed by its own bus and can be read or written on every clock cycle.

- **Deep Pipelining.** Normally, DSPs implement a highly pipelined design that leads to faster code execution and a more efficient use of the processor resources.

- **Dedicated address generation units.** Memory accesses in digital signal processing algorithms tend to exhibit very predictable patterns [11]. Therefore, DSPs implement dedicated address generation units that take advantage of this predictability by supporting specialised addressing modes like circular addressing and bit-reversed addressing.
Chapter 3: Background on Digital Control II. Controller Implementation

- **Extended Parallelism - SIMD, VLIW and superscalar processing.** The trend in DSP architecture design is to increase both the number of instructions executed in each cycle and the number of operations performed per instruction to enhance performance ([35], [28]). In newer DSP architectures, parallel processing techniques such as SIMD, VLIW and static superscalar, are extensively used to achieve increased computational performance [26]. The main characteristics of these techniques were reviewed in Section 3.3.3.

- **Special instructions dedicated to digital signal processing.** An optimised hardware architecture must be supported by a particular set of instructions that take advantage of the new design features. The benefits of these special instructions can be twofold: they lead to more compact code that takes up less space in memory, and they lead to an increase in the execution speed of digital signal processing algorithms.

- **Specialised ALU.** As shown before, in order to make real-time signal processing feasible, DSPs require special ALU architectures. They normally implement optimised MAC units and support SIMD operations. Fixed-point arithmetic is widely implemented while certain families include optimised floating-point units.

- **Efficient looping, low interrupt overheads, dedicated Input/Output (serial or parallel interfaces)[6].**

Since their introduction in the early 1980s, DSPs have grown substantially in complexity and sophistication to enhance their capabilities and range of applicability. For convenience, DSP processors can be divided into two broad categories [26]:

- **General-purpose DSPs.** They are basically high speed microprocessors with hardware architectures and instruction sets optimised for digital signals processing operations.

- **Special-purpose DSPs.** They are tailored to execute specific digital signal processing algorithms, for example FIR filters, or for the efficient execution of some application-dependent operations.
In wide bandwidth applications where the input/output data rates are high, most general-purpose DSPs cannot perform the required computations fast enough and special-purpose DSPs are sought. Furthermore, for a given applications, most general-purpose DSPs contain many on-chip resources that are either redundant or underutilised. On the other hand, special-purpose DSPs offer speed but lack the flexibility of general-purpose DSPs. In general, if a single-chip special-purpose DSP exists for a particular task, it is the preferred option because it has lower gate count, does not require knowledge of assembly language and does not have problems of software debugging.

There are a number of reasons why DSPs are still the solution of choice for many applications. Although other types of processors may provide similar (or better) execution speed for some demanding real-time applications, DSPs often provide a better mixture of performance, power consumption, and price than the rest of off-the-shelf devices. Another key advantage is the availability of DSP-specific development tools and off-the-shelf DSP software components.

3.6.3 Microcontroller Unit (MCU)

MCUs are systems that can operate on their own in a stand-alone fashion offering an inexpensive low power embedded control and interfacing solution with reasonable performance. Its design emphasises self-sufficiency and cost-effectiveness, in contrast to a general-purpose microprocessor. They include all the necessary components of a complete computer system on a single chip (e.g. clock system, CPU, RAM memory, programmable input/output ports) and some specific components like AD/DA converters, timer/counters and some sort of non-volatile memory for program storage (e.g. FLASH). The instruction set of MCUs are in general more restricted than in embedded off-the-shelf microprocessors and memory may often be quite limited. Usually microcontrollers are offered as a family, different members of the family having extra in-built facilities [12] and varying their key parameters (e.g. program memory, SRAM, I/O pins, etc). Traditionally, 8-bit families have dominated the market, based on well known and reliable 8-bit processor architectures such as Intel's 8051 [37], Zilog's Z80
Chapter 3: Background on Digital Control II. Controller Implementation

[52] and Motorola's 68HC08. However, depending on the application and the budget 16-bit families are preferred. New technologies and applications had led to the development of new 32-bit families, based on modern 32-bit RISC processors. Performancewise, a MCU is limited by the speed of its CPU, its (usually) shared internal bus (von-Neumann architecture), its register file size and its ALU capabilities amongst others. It is important to notice that such devices are usually intended as control logic in low-power and/or device interfacing applications. Therefore, they are usually not an option in case of real-time requirements at high sample rates.

Texas Instrument's MSP430 family [29] is a current good example that summarises the above described MCU characteristics. Its ultra-low-power consumption architecture is intended for battery-based use. It implements a single cycle 16-bit RISC CPU with a reduced instruction set of 27 instruction on a von-Neumann architecture. Its design includes specific components like AD/DA converters, comparators, operational amplifiers, SRAM and FLASH amongst others. The MSP430 family consist of around 60 devices ranging in price approximately from $0.5 to $10 (for a suggested 1,000 parts resale in US, 2004). Its key applications are: utility metering, portable instrumentation and intelligent sensing.

3.6.4 Application Specific Processor

In almost any application, special-purpose processors provide better performance that general architectures. Their fit-to-purpose architecture tends to balance the performance and constraints required in their particular field. By exploring the particular arithmetic and computational needs of the algorithm to be implemented on them, they can match its structure and not the other way round, implementing only the resources that such algorithms require.

On the other hand, application specific architectures are not free of drawbacks. One is the time required to design, develop and test the processor's hardware and its software tools. Another is the risk of developing an overspecialised architecture. However, once a suitable application specific processor is developed it is likely to deliver significantly better performance (for a particular type of application) than any other
This chapter reviewed the main numerical and implementation requirements common to digital signal processing algorithms. The major embedded processing architectures used for executing such algorithms were presented and their advantages and shortcomings were identified. In order to understand the architectural decisions taken in the design of the application specific processor presented in Chapter 5 a background knowledge in computer architecture for signal processing was provided. Next, the key points introduced in this chapter are summarised.

Implementation requirements

- **Arithmetic characteristics.** Certain arithmetic characteristics were found common to most digital signal processing implementations, that led to the following conclusions: 1) The arithmetic unit must be able to execute at least one MAC instruction per clock cycle. 2) The arithmetic format to be used for implementing the coefficients and internal variables is one of the basic implementation decision for it will have an impact in the design speed, sample rates, precision, *signal to noise ratio* (SNR), etc. 3) If variables are represented in fixed-point format, saturation logic must be implemented at the controller's output and the variables should be extended with a number of overflow bits.

- **I/O Ports.** I/O Ports provide a means of exchanging data with external devices. The use of standard communication channels (e.g. AMBA, UART, USB, etc.) assure seamless SoC connectivity since they are supported by most major vendors. AD/DA conversion is very often required and if supported by the processing architecture it could improve performance and minimise the external logic requirements.

- **Embedded architecture for real-time performance.** In the embedded space, real-time performance for digital signal processing requires an optimised architecture.
different from the one found in general purpose processors. This architecture should implement a very high bandwidth memory system that supports several memory accesses per clock cycle and optimised data paths for intensive arithmetic operations' execution. Unit replication, if implemented, can lead into parallel execution of instructions. The execution overheads must be reduced to improve sample-rates and the design must praise predictability.

**Hardware Architecture for Signal Processing**

The following list summarises those characteristics expected to be found in a real-time signal processing architecture:

- **Instruction Set Architecture.** Storage type based on a register-to-register (load-store) architecture. Support for addressing modes like displacement, immediate and register indirect. A reduction in the specialised DSP addressing modes due to the programmer's increasing rely on compilers. Overflow detection (in the case of fixed-point arithmetic) and saturation logic should be implemented. Basic operations usually supported: load, store, add, subtract, move register, register and shift. MAC operations must be supported and optimised through specialised MAC units. Loop overheads should be optimised. Instruction Set preferred encoding: variable length for improving code size, or fixed length for improving performance. It is important to remark the lack of compilers' support for vector instructions and specialised addressing modes.

- **Implementations.** Harvard Architecture for simultaneous access to data and code. Multiple on-chip memory blocks or registers to store internal variables and coefficients. Pipelined architecture with single-cycle MAC instruction.

- **Parallelism.** Single Issue architectures: Pipelining and SIMD. Multi-Issue architectures: VLIW and superscalar. VLIW architectures provide poor code density and rely on the compiler and the programmer to obtain a reasonable increase in the performance with respect to the single-issue counterpart. Superscalar architectures on the other hand are too demanding in terms of gates and power and introduce a certain level of unpredictability in the algorithm execution.
Chapter 3: Background on Digital Control II. Controller Implementation

An important point to remember is the big gap between the hardware and the software generated by the compiler.

Therefore, it can be concluded that in the particular field of digital control and filtering, it would be highly desirable to have a hardware architecture optimised for real-time control and supported by software tools that efficiently translated the desired control algorithm into the processors' native language, using all the resources that this processing architecture could offer.

Devices and Architectures

- **General Purpose Microprocessors.** They are powerful and fast, and recently their architecture has been enriched with signal processing capabilities. However, their size, power consumption and generality makes them unsuitable for embedded applications.

- **MCUs.** They are a single-chip solution primarily targeted to provide low power, inexpensive, programmable logic control and interfacing to external devices. They usually incorporate a simplified CPU with a reduced fit-to-purpose instruction set, memory, programmable input/output ports, timers/counters, AD/DC converters and serial units. They fulfil the embedded requirements but their architecture is not designed for high demanding real-time performance.

- **DSPs.** They are usually the architecture of choice in high-performance digital signal processing applications because they offer the best balance between performance, size and power consumption among the traditional off-the-shelf devices. However, their designs are growing in complexity and generality, increasing their size and power consumption. Compilers don't usually generate an assembly code that make the most of these new architectural improvements. As real-time applications increasingly demand more computational power and higher sampling rates, most DSPs still require handwritten assembler routines in order to meet the required performance.

- **Application Specific Processors.** If they exist for a given application, these targeted architectures usually offer the best performance and most efficient resource
utilisation. If they don’t exist and must be created from scratch, their design, development and testing time must be considered as time-to-market may be significantly delayed (depending on the complexity of the required architecture). However, very high-performance applications may only be feasibly addressed by means of targeted architectures.
Chapter 4

Delta Formulation and Floating-Point Implementation

4.1 Introduction and Objectives of the Chapter

Controller formulations based on the $\delta$-operator are acknowledged to overcome, in many cases, the numerical problems that the shift operator introduces when implemented with finite word length precision. The numerical benefits of the delta formulation also reduce the dynamic range of the control coefficients allowing them to be represented with a smaller number of bits. Nevertheless, its use within the control and signal processing community is rare and often its existence is unknown. Furthermore, there is a minimal amount of literature available that covers the delta formulation, its teaching is uncommon and even control and signal processing related software tools such as MATLAB don't implement it.

Floating-point architectures provide wider dynamic ranges and higher accuracy levels than their fixed-point counterparts, at the cost of more complex hardware implementations. This added complexity implies bigger silicon area, more power consumption and slower clock rates (for a given implementation technology). Although the IEEE 754 Standard format is one of the most widely used binary floating-point systems, in many applications its dynamic range and accuracy are not required. In those cases, an arithmetic unit based on a shorter word length floating-point archi-
tecture would balance complexity, speed and performance, providing a well-suited fit-to-purpose solution.

This chapter will present the foundations of the $\delta$-operator canonic representation, which combined with a reduced word length floating-point formulation will lead to the design of the numerically robust and highly optimised arithmetic unit architecture presented in Chapter 5.

The objectives of the chapter are to:

- Present the $\delta$-operator state-space canonic and modified formulation, whose numerical properties are essential to validate the use of reduced word length floating-point coefficients.
- Obtain a custom floating-point representation that can lead to a fit-to-purpose arithmetic unit architecture tailored for real-time embedded control.
- Analyse the floating-point formulation used in the implementation of the rCSP's arithmetic unit.

### 4.2 Delta-based State-Space Formulation

In this section the structure and equations of the canonic and modified canonic delta form shall be derived from a general transfer function in $\gamma$, the Delta-Transform variable (see Appendix F).

#### 4.2.1 Canonic Delta Structure

Let $F(\gamma)$ be a general transfer function expressed in negative powers of $\gamma$.

$$F(\gamma) = \frac{b_m \gamma^{-m} + b_{m-1} \gamma^{-(m+1)} + \ldots + b_1 \gamma^{-1} + b_0}{a_n \gamma^{-n} + a_{n-1} \gamma^{-(n+1)} + \ldots + a_1 \gamma^{-1} + a_0}$$

Without losing generality two assumptions are made:

1. $a_0 = 1$ (i.e. equivalent to divide numerator and denominator by $a_0$).
2. \( m = n = \text{max}\{n, m\} \)

In order to avoid the use of double indexing, numerator and denominator order will be equalised so \( m = n = \text{max}\{n, m\} \) (i.e. equivalent to include some extra null coefficients).

Then, Equation (4.1) can be expressed as:

\[
F(\gamma) = \frac{b_n \gamma^{-n} + b_{n-1} \gamma^{-(n+1)} + \ldots + b_1 \gamma^{-1} + b_0}{a_n \gamma^{-n} + a_{n-1} \gamma^{-(n+1)} + \ldots + a_1 \gamma^{-1} + 1}
\]

(4.2)

The transfer function in Equation (4.2) can be represented as

\[
F(\gamma) = \frac{Y(\gamma)}{U(\gamma)} = \frac{V(\gamma) H(\gamma)}{H(\gamma) U(\gamma)}
\]

(4.3)

leading to the decomposition

\[
\begin{align*}
Y(\gamma) &= b_n \gamma^{-n} + b_{n-1} \gamma^{-(n+1)} + \ldots + b_1 \gamma^{-1} + b_0 \\
H(\gamma) &= 1 \\
U(\gamma) &= a_n \gamma^{-n} + a_{n-1} \gamma^{-(n+1)} + \ldots + a_1 \gamma^{-1} + 1
\end{align*}
\]

(4.4) \hspace{1cm} (4.5)

Reordering terms in Equations (4.4) and (4.5) it can be readily seen that

\[
\begin{align*}
Y(\gamma) &= b_n \gamma^{-n} H(\gamma) + b_{n-1} \gamma^{-(n+1)} H(\gamma) + \ldots + b_1 \gamma^{-1} H(\gamma) + b_0 H(\gamma) \\
H(\gamma) &= U(\gamma) - a_n \gamma^{-n} H(\gamma) - a_{n-1} \gamma^{-(n+1)} H(\gamma) - \ldots - a_1 \gamma^{-1} H(\gamma)
\end{align*}
\]

(4.6) \hspace{1cm} (4.7)

The following coefficients’ substitution will lead to a simplified analysis:

\[
\begin{align*}
b'_i &= b_{n-i+1}, \quad i = 1, 2, \ldots n \\
a'_i &= a_{n-i+1}, \quad i = 1, 2, \ldots n
\end{align*}
\]

(4.8) \hspace{1cm} (4.9)

This is equivalent to transpose the coefficients (so, for instance \( b'_1 \equiv b_n \), \( b'_2 \equiv b_{n-1} \) or \( b'_n \equiv b_1 \)), thus naturally operating with a transposed set of state variables. In that way, the final state space representation is directly obtained from the following analysis. Otherwise, the resultant state space formulation should be transposed at the end in order to obtain a more suitable set of real-time implementation equations.

Substituting the new coefficients into Equations (4.6) and (4.7) leads to the construction of the delta canonical block diagram shown in Figure 4.1. From this figure the relation between contiguous state variables is calculated as:

\[
X_i(\gamma) = \gamma^{-1} X_{i+1}(\gamma), \quad i = 1, 2, \ldots, n
\]

(4.10)
As demonstrated in [39], for null initial conditions it is possible to move from the Delta-Transform domain to the delta form of difference equations just by replacing $\gamma$ with $\delta$ (and vice versa). Thus, $\gamma$ and $\delta$ are linked in the same way that $z$ and $q$ are. This substitution is applied in Equation (4.11) to calculate the state variables discrete evolution in terms of the $\delta$-operator:

$$\delta x_i[k] = x_{i+1}[k], \ i = 1, 2, ..., n$$  \hspace{1cm} (4.12)\

and applying the delta operator as defined in Equation (2.8)

$$x_i[k + 1] = x_i[k] + x_{i+1}[k], \ i = 1, 2, ..., n$$  \hspace{1cm} (4.13)

For clarity, $x_{n+1}$ is rename as $h$ (since $x_i$ will represent the state variables). Then

$$\delta^{-1}h[k] = x_n[k]$$

and therefore

$$x_n[k + 1] = x_n[k] + h[k]$$  \hspace{1cm} (4.14)\

From the block diagram in Figure (4.1) it can be readily seen that

$$h[k] = u[k] - \sum_{i=1}^{n} a'_i x_i$$  \hspace{1cm} (4.15)
From Equations (4.13), (4.14) and (4.15) the following state equations are obtained:

\[
\begin{align*}
    x_1[k+1] & = x_1[k] + x_2[k] \\
    x_2[k+1] & = x_2[k] + x_3[k] \\
    & \vdots \\
    x_{n-1}[k+1] & = x_{n-1}[k] + x_n[k] \\
    x_n[k+1] & = (1 - a'_n)x_n[k] - \sum_{i=1}^{n-1} a'_i x_i + u[k]
\end{align*}
\] (4.16) (4.17) (4.18) (4.19) (4.20)

In the same way \(y[k]\) can be obtained from Figure 4.1 by noticing that:

\[
y[k] = b_0 y_0[k] + \sum_{i=1}^{n} b'_i x_i
\] (4.21)

and then substituting Equation (4.15) into Equation (4.21), leading to the expression:

\[
y[k] = b_0 u[k] - b_0 \sum_{i=1}^{n} a'_i x_i + \sum_{i=1}^{n} b'_i x_i
\]

Finally, reordering terms \(y[k]\) can be expressed as:

\[
y[k] = b_0 u[k] + \sum_{i=1}^{n} (b'_i - b_0 a'_i)x_i
\] (4.22)

Thus, the following canonic delta state space representation of the initial transfer function is obtained by expressing Equations (4.16) to (4.20) and Equation (4.22) in a matrix formulation and restoring the initial set of coefficients \(\{a_i, b_i\}\):

\[
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}_{k+1}
= 
\begin{pmatrix}
    1 & 1 & 0 & \ldots & 0 \\
    0 & \ddots & \ddots & \ddots & \vdots \\
    0 & 0 & \ddots & \ddots & 0 \\
    \vdots & \vdots & \ddots & 1 & 1 \\
    -a_n & -a_{n-1} & \ldots & -a_2 & (1 - a_1)
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}_k
+ 
\begin{pmatrix}
    0 \\
    0 \\
    \vdots \\
    1
\end{pmatrix}
u_k
\] (4.23)

\[
y_k = 
\begin{pmatrix}
    b_n - a_n b_0 |b_{n-1} - a_{n-1} b_0| \ldots |b_1 - a_1 b_0|
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}_k
+ 
b_0 \cdot u_k
\] (4.24)
4.2.2 Modified Canonic Delta Structure

In this section, the canonical delta block diagram shown in Figure 4.1 shall be modified, reallocating the denominator's coefficients into the delta path and correcting accordingly the numerator's coefficients, thus obtaining a better scaled set of coefficients. This reduces the dynamic range that needs to be supported by the floating-point system in order to represent the different coefficients, thus leading to a smaller word length arithmetic representation. The reason is that the original denominator coefficients \( \{a_i\} \) in the canonic form are of the order \( T^i \) (where \( T \) is the sample period), and therefore they are progressively small as \( i \) increases. On the other hand, if these coefficients are moved into the forward path, the new set of reallocated coefficients \( \{q_n\} \) become of order \( T \). Further details can be found in Appendix F.

The reallocation of coefficients into the delta path is exemplified in Figure 4.2. As can be readily seen, the reallocation of the denominator's coefficient \( a'_i \) has an immediate effect on the system that should be compensated by replacing coefficient \( b'_i \) with \( (b'_i/a'_i) \equiv p_i \). In the same way, the next denominator coefficient \( (a'_{i-1}) \) that is moved into the delta path should be also compensated, becoming \( (a'_{i-1}/a'_i) \equiv q_{i-1} \). Note that \( q_i \) represents a number (and therefore it does not represent the shift operator). In this chapter the shift operator is not used. Note also that the shift operator would not be indexed.

By iterating this process throughout the whole set of denominator's coefficients the block diagram for the transfer function \( F(\gamma) \) in Modified Canonic Delta form is obtained (see Figure 4.3).
Chapter 4: Delta Formulation and Floating-Point Implementation

Figure 4.3: Block diagram representation of $F(\gamma)$ in Modified Canonic Delta Form

Analysing two contiguous state variables:

$$X_i(\gamma) = q_i\gamma^{-1}X_{i+1}(\gamma) \quad (4.25)$$

so

$$\gamma X_i(\gamma) = q_iX_{i+1}(\gamma) \quad (4.26)$$

which in the $\delta$-operator form can be expressed as

$$\delta x_i[k] = q_i x_{i+1}[k] \quad (4.27)$$

Applying the definition of the $\delta$-operator to Equation (4.27) leads to the general discrete expression of the state variables:

$$x_i[k+1] = x_i[k] + q_i x_{i+1}[k] \quad , i = 1, 2, \ldots, n \quad (4.28)$$

For the particular case of $x_n[k+1]$ (renaming again $x_{n+1}$ as $h$):

$$x_n[k+1] = x_n[k] + q_n h[k] \quad (4.29)$$

where $h[k]$ can be obtained as:

$$h[k] = u[k] - \sum_{i=1}^{n} x_i \quad (4.30)$$

From Equations (4.28) to (4.30) the following state equations are obtained:

$$x_1[k+1] = x_1[k] + q_1 x_2[k] \quad (4.31)$$

$$x_2[k+1] = x_2[k] + q_2 x_3[k] \quad (4.32)$$
Chapter 4: Delta Formulation and Floating-Point Implementation

\[ x_{n-1}[k + 1] = x_{n-1}[k] + q_{n-1}x_n[k] \]

\[ x_n[k + 1] = (1 - q_n)x_n[k] - q_n \sum_{i=1}^{n-1} x_i + q_n u[k] \]

In the same way, \( y[k] \) can be obtained from Figure 4.3, noticing that

\[ y[k] = p_0 h[k] + \sum_{i=1}^{n} p_i x_i \]

Substituting Equation (4.30) into Equation (4.36)

\[ y[k] = p_0 u[k] - p_0 \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} p_i x_i \]

and reordering terms, the state equation of the output is obtained:

\[ y[k] = \sum_{i=1}^{n} (p_i - p_0) x_i + p_0 u[k] \]

Finally, representing in a matrix formulation the previously obtained state equations and output the modified canonic state space representation in delta is obtained:

\[
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}_{k+1} =
\begin{pmatrix}
    1 & q_1 & 0 & \cdots & 0 \\
    0 & 1 & q_2 & \ddots & 0 \\
    \vdots & \ddots & \ddots & \ddots & 0 \\
    0 & \cdots & 0 & 1 & q_{n-1} \\
    -q_n & -q_n & \cdots & -q_n & (1 - q_n)
\end{pmatrix}_{n \times n}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}_k +
\begin{pmatrix}
    0 \\
    0 \\
    \vdots \\
    q_n
\end{pmatrix} + p_0 \cdot u_k \tag{4.38}
\]

\[
y_k = \begin{pmatrix}
p_1 - p_0 \\
p_2 - p_0 \\
\vdots \\
p_n - p_0
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{pmatrix}_{k} + p_0 \cdot u_k \tag{4.39}
\]

where

\[ p_i = \frac{b_i}{a_i}, \quad q_i = \frac{a_i}{a_{i+1}} \quad \forall i = 1, 2, \ldots, n \]

\[ p_0 = b_0, \]

\[ q_n = a_1 \]
4.2.3 Real-time Implementation

The real-time equations derived from the modified canonic delta formulation of an n-th order SISO controller are presented next, embedded in a typical DSP and MCU control loop implementation.

\[
c_1 = p_1 - p_0; \quad c_2 = p_2 - p_0; \quad \ldots; \quad c_n = p_n - p_0;
\]

while (control_loop_enabled) {
    // Reading input \( u[k] \)
    \( u = \text{readInput}(i); \)
    // Action \( y[k] \) calculation and outputting
    \( y = c_1x_1 + c_2x_2 + \ldots + c_nx_n + p_0U; \)
    sendAction(y);
    // Calculating the contribution of the \( X[k] \) for \( x_n[k+1] \)
    \( x_n_t = x_n - q_nx_1 - q_nx_2 - \ldots - q_nx_n; \)
    // Updating the state variables \( X[k+1] \)
    \( x_1 = x_1 + q_1x_2; \)
    \( x_2 = x_2 + q_2x_3; \)
    \( \ldots \)
    \( x_n = x_n_t + q_nu; \)
    // Wait for synchronisation
    waitSynch(s_time);
}

As can be appreciated, the canonic component \( 1 - q_n \) is decomposed in order to calculate \( x_n[k+1] \). This prevents unevenly distributed quantisation errors affecting the steady-state response of the system. The reason is that since \( q_n \equiv a_n \) tend to zero as the sample rate increases (see Appendix F), the factor \( 1 - q_n \) would tend to one. As the Section 4.3 will show, in sort word length floating-point formulations the difference between the maximum quantisation error in the vicinity of zero and in the vicinity of one could be relevant enough not to be neglected.

rCSP-based control implementations present an optimised structure of the previously presented control loop. As Chapter 5 will show, the rCSP eliminates any loop overhead, branching or function call. In the same way, its architecture provides a single
cycle instruction for outputting the data (in either digital or modulated form), and it does not implement a readInput function nor instruction since the sampling process and the digital data access is performed in parallel with the control loop execution. Therefore, sampled and digital data is accessed and processed at the same time in a single cycle arithmetic instruction.

4.3 Reduced Word Length Floating-Point Implementations

Floating-point numbers are digital representations of a finite subset of the real numbers. The range of representable numbers that a particular floating-point system offers is directly related to its word length. Within this range only a small proportion of numbers are accurately represented, the rest being approximated by the nearest number in the subset. In general, a real number \( \nu_r \) can be approximated by a floating-point number \( \nu \) as:

\[
\nu_r \simeq \nu = m \cdot b^e
\]  

(4.40)

where \( m \) is regarded as significant or mantissa, \( b \) is the base or the radix, and \( e \) is the exponent[26]. Normally, in engineering systems the selected base is 2. There are different ways of calculating the decimal value of a floating-point number, depending on the way the mantissa and exponent are defined. Often, the exponent is an unsigned integer biased by certain amount, being able in that way to represent negative numbers without using two's complement. This means that a fixed value is subtracted from the exponent field to get the true exponent value. Thus, a floating-point number \( \nu \) with its exponent biased by a fixed amount \( \text{bias} \) can be represented as:

\[
\nu = m \cdot 2^{e-\text{bias}} = m \cdot 2^{be}
\]  

(4.41)

where \( be \) is the biased exponent. The mantissa can be expressed in two's complement or with a separate sign bit and usually is normalised to be between 0 and 1. In addition, floating-point representations often include the special values \( +\infty, -\infty \) (positive and negative infinity), and NaN ('Not a Number'). Infinities are used when results are too large to be represented, and NaNs indicate an invalid operation or undefined result.
Chapter 4: Delta Formulation and Floating-Point Implementation

One of the most widely used binary floating-point systems is the IEEE 754 Standard [22]. In its single precision representations (see Figure 4.4), 8 bits are used for encoding the exponent, 23 bits for encoding the mantissa and the most significant bit is used as sign bit. The decimal value of a normalised IEEE floating-point number is given by Equation (4.42) [26]

\[ \nu = (-1)^s(1 \cdot F)2^{e-127} \]  

(4.42)

where \( F \) is the mantissa in 2's complement binary fraction represented by bits 0 to 22, the exponent is biased 127 units and the sign bit \( s \) is 0 for positive numbers and -1 for negative ones.

![IEEE single precision floating-point representation](image)

Figure 4.4: IEEE single precision floating-point representation

Although a floating-point representation provides in general much wider dynamic range and precision than a fixed-point one, its resolution is not evenly distributed along the whole range of numbers that can be represented (see Figure 4.5). This fact, generally disregarded when using a standard IEEE representation, can become an issue in shorter word length floating-point formulations. Digital controllers formulated with \( \delta \)-operator usually have most of their coefficients in the vicinity of zero, while coefficients greater than one are mainly used for representing gains. This suggests that decreasing the range of representable numbers while concentrating the resolution around zero would produce a numerically valid floating-point formulation that would lead to a simpler, smaller and faster fit-to-purpose arithmetic unit architecture. Thus, in many control applications, especially if the controller is formulated in the delta form, the 32 bits control coefficients’ word length required by the IEEE 754 Standard floating-point representation is not only unnecessary but also a source of extra hardware complexity and computational overheads.
4.3.1 Floating-point formulation of the rCSP processor

The rCSP’s arithmetic unit architecture is described in detail in Chapter 5, but a summary of the coefficient’s representation in the processor’s arithmetic unit standard configuration (see Figure 5.3) is provided in Table 4.1. In general, the range of mantissa, exponent, and bias exponent can be calculated as:

\[ m \in [-2^{ml-1}, 2^{ml-1} - 1] \quad (4.43) \]
\[ e \in [0, 2^{el} - 1] \quad (4.44) \]
\[ be \in [-bias, (2^{el} - 1) - bias] \quad (4.45) \]

and the range of representable numbers can be expressed as:

\[ \nu \in [m_{\text{min}} \cdot e_{\text{max}}, m_{\text{max}} \cdot e_{\text{max}}] \quad (4.46) \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Length (bits)</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mantissa (m)</td>
<td>((ml) = 6)</td>
<td>Two's complement</td>
</tr>
<tr>
<td>exponent (e)</td>
<td>((el) = 5)</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>COEFFICIENT</td>
<td>11</td>
<td>Custom floating-point</td>
</tr>
</tbody>
</table>

Table 4.1: Coefficients’ parameters in the rCSP’s standard configuration

Standard bias definitions usually focus on providing a wide range of representable numbers. However, as stated before, delta formulated controllers tend to have most of their coefficients in the vicinity of zero. In order to concentrate the resolution distribution around zero while allowing for significantly greater than unity gains, an
alternative bias is needed. The following bias definition

\[ bias = 2^e - 2 \]  

(4.47)

provides a biased exponent (be) range of:

\[ be \in [-bias, 1] \]  

(4.48)

Thus, the rCSP's standard configuration provides the following ranges for the coefficient's parameters:

\[ m \in [-32, 31] \]  

(4.49)

\[ e \in [0, 31] \]  

(4.50)

\[ be \in [-30, 1] \]  

(4.51)

with a range of numbers

\[ \nu \in [-64, 62] \]  

(4.52)

that allows to represent gains significantly bigger than one, yet concentrating the resolution around zero. This higher resolution around zero is graphically depicted in Figure 4.6, which shows that 91% of the coefficients that are representable between 0 and 1 are located in the range \( \nu \in [0, 0.1) \). This is an important result, since most of the coefficients (in a delta formulation) tend to move towards zero as the sample rates increase (see Appendix F).

Figure 4.6: Graphical representation of the resolution distribution for \( \nu \in [0, 1] \)
If \( \nu \) is a real number as defined in Equation (4.40) and \( \nu \) is the approximated floating-point number as defined in Equation (4.41), then the maximum quantisation error can be calculated as:

\[
\text{error}(\nu) = \frac{(m + 1) \cdot 2^{b_e} - m \cdot 2^{b_e}}{2} = 2^{b_e - 1} = \frac{\text{resolution}}{2} \quad (4.53)
\]

for \( m \in [2^{m_l - 2}, 2^{m_l - 1} - 1] \). The quantisation error percentage only depends on the mantissa as the following equation shows:

\[
\%\text{error}(\nu) = \frac{\text{error}(\nu)}{m \cdot 2^{b_e} + \text{error}(\nu)} \cdot 100\% = \frac{1}{2m + 1} \cdot 100 \quad (4.54)
\]

Therefore, the maximum quantisation error percentage is found when \( m = 2^{m_l - 2} \), and its value is:

\[
\text{error}_{\text{max}}\% = \frac{1}{2^{m_l - 1} + 1} \cdot 100\% \quad (4.55)
\]

which for the rCSP's standard arithmetic unit configuration corresponds to:

\[
\text{error}_{\text{max}}\% = \frac{1}{2^{6} + 1} \cdot 100\% = 3.03\% \quad (4.56)
\]

### 4.4 Summary

The first section of this chapter introduced a state-space controller formulation based on the \( \delta \)-operator, which is numerically more robust than the traditional shift-operator formulation, especially at high-sample rates, hence being particularly suitable for real-time systems, especially in its modified canonic form. The well-scaled nature of its coefficients allows them to be represented using a short word length floating-point formulation, opening the possibility of more efficient arithmetic units for control that are smaller in size and power consumption, yet high in performance.

Afterwards, the basics of floating-point representation were introduced, analysing its short word length numerical properties. The numerical characteristics of the rCSP's standard arithmetic unit configuration were then obtained.

This chapter demonstrated the feasibility of a targeted arithmetic unit design with a reduced word length floating-point representation. Such arithmetic unit would be especially tailored for embedded real-time oversampled systems since its size, complexity
and power consumption would be decreased (essential for embedded systems) but its fit-to-purpose design would allow a high-performance arithmetic execution (essential for demanding real-time systems).

The floating-point formulation used in the rCSP processor for representing the coefficients (see Chapter 5) is based on the results of this chapter.
Chapter 5

Reconfigurable Control System Processor (rCSP) Architecture

5.1 Introduction and objectives of the chapter

Application Specific Processors (ASP) are specialised architectures able to perform a particular set of tasks more efficiently than common general purpose devices. Such tasks must be carefully defined since overspecialisation (i.e., an over-restricted application's scope) will lead to the design of an architecture which lacks the flexibility to cover a wide range of applications with similar (but not equal) characteristics. On the other hand, an excess in generality will inherit some of the drawbacks typical to general devices. As it has been shown in Section 3.6, there is a number of off-the-shelf architectures already available in the market that are commonly used for real-time control and/or signal processing. However, these architectures present certain drawbacks such as an excess of generality (that makes them unsuitable for example for resource-aware embedded applications) or a performance level insufficient to address real-time applications with high sample rates.

The rCSP philosophy differs in several aspects from them. First, its architecture doesn't come from a simplification of an embedded CPU or any other device. It comes from the analysis of the requirements found in control and signal processing algorithms optimised for execution under real-time performance in embedded devices.
Then, the general structure of such algorithms has been mapped into hardware. This approach has been used recently by the digital signal processing industry for increasing the performance of particular algorithms, resulting in efficient but too specialised solutions (e.g. FFT processors and IIR filters). In fact, the rCSP could be related to the application specific DSP families, although its architectural philosophy as a reconfigurable soft IP core, its design methodology and software tools, and its seamless System-on-Chip integration make it significantly different from its DSP relatives. Thus, the application specific processor described in this chapter combines the efficient execution of targeted architectures with the flexibility of programmable and reconfigurable architectures, offering a fit-to-purpose yet flexible solution to the problem of real-time embedded control implementations. Due to the processor's customisable architecture, its standard configuration will be used in this chapter to describe its design and implementation.

The objectives of this chapter are to:

- Introduce the rCSP processor as a reconfigurable fit-to-purpose processor for high-performance real-time embedded control applications.

- Give a detailed description of the processor's instruction set architecture.

- Provide an in-depth overview of the different processor's functional modules and how are they implemented.

- Cover the pipeline and data path structure of the processor, enumerating the possible pipeline hazards and how are they addressed in hardware.

- Describe how to interface with the processor through its integrated Advanced Microcontroller Bus Architecture (AMBA) interface.

- Present the analogue-to-digital and digital-to-analogue conversion units integrated into the processor's core.

- Show how to configure the rCSP's architecture.

- Briefly describe the possibility of building multi-rCSP parallel architectures.

- Provide speed and complexity figures for the processor's implementation.
5.2 Architecture I: Overview

The rCSP is a five stage pipelined reconfigurable state-space processor with a reduced and optimised fixed-length RISC like 32-bit\(^1\) instruction set architecture. All its instructions are executed in a single clock cycle (including MAC operations), its hardware being in charge of handling all the possible pipeline hazards produced by instruction dependencies. It implements a modified Harvard architecture with four different register banks for variables and coefficients, and a separate memory bank for instructions. The rCSP architecture is optimised for real-time embedded MIMO control law execution and digital filtering, covering both LTI and LTV systems. The latter is addressed by means of its built-in real-time adaptation capabilities. In order to get the highest performance without relying on the particular hardware and programming skills of the engineer, its design allows a fully predictable execution, implements reduced and specialised instruction set architecture, allows seamless communication with other digital modules and integrates AD/DA converter logic on a single chip. Its design (developed as a soft IP core in VHDL) synthesises for both FPGA and ASIC technologies and is not restricted to a particular vendor. This is particularly useful in the case of FPGA implementations, where the design can take advantage of the constant improvements in this semiconductor technology, with the added advantage of its reconfigurability, fast prototyping and straight-forward final implementation. Specially designed for the embedded space, its architecture is reconfigurable to match the particular application’s requirements. This allows no waste of hardware resources (e.g. memory, gates, routing space) and therefore a lower power consumption. The processor can be reprogrammed in real-time by other devices, a feature that proves to be useful, for example in fault tolerant architectures. The processor can also run in fault detection mode where a known control law is executed and compared against a known correct output. This mode can be used for instance at booting time to check the processor’s integrity in particularly hazardous environments. The rCSP’s outputs can be transferred to another digital module and/or as a modulated action to analogue actuators. Input signals can be brought into the processor either as digital inputs or

\(^1\)32-bit instruction size corresponds to the default standard configuration. Reconfigurability can optimise the instruction size according to the memory allocation requirement.
analogue signals. The communication with external modules is accomplished through its built-in Advanced Microcontroller Bus Architecture (AMBA) interface. Through this interface the rCSP can be commanded, remotely debugged, monitored in real-time, its coefficients adapted, and it can exchange data with a variety of modules (e.g. embedded processors).

In conclusion, the processor's instruction set and architecture is tailored to address real-time embedded control applications with high sample rates. Its scalable design evolved from the analysis of the particular requirements found in digital control applications (see Section 3.2) its main characteristics being summarised in Table 5.1.

<table>
<thead>
<tr>
<th>real-time embedded scalability</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓    -  -</td>
<td>Optimised data path for signal processing</td>
</tr>
<tr>
<td>✓    -  -</td>
<td>One cycle instruction execution</td>
</tr>
<tr>
<td>✓    -  -</td>
<td>One cycle MAC instruction execution</td>
</tr>
<tr>
<td>✓    -  -</td>
<td>Predictability in the execution</td>
</tr>
<tr>
<td>✓    ✓  -</td>
<td>Reduced instruction set optimised for control</td>
</tr>
<tr>
<td>-    ✓  -</td>
<td>Integrated AD/DA logic</td>
</tr>
<tr>
<td>-    ✓  -</td>
<td>Seamless interfacing with external devices</td>
</tr>
<tr>
<td>-    ✓  -</td>
<td>Reprogrammable in real-time</td>
</tr>
<tr>
<td>-    ✓  ✓</td>
<td>Reconfigurable architecture</td>
</tr>
<tr>
<td>-    -  ✓</td>
<td>Scalable architecture</td>
</tr>
<tr>
<td>-    -  ✓</td>
<td>Scalable architecture</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of digital control requirements covered by the rCSP processor

5.3 Architecture II: Instruction Set Architecture

As stated before, the instruction set architecture refers to the portion of the processor visible to the programmer or compiler writer. In the case of the rCSP, the programmer doesn't need to get concerned with the processors' programming language because its cross-compiler (rcspcc.m) accepts control systems descriptions and equa-
tions as source code. The software tools that allow the automation of the system’s programming as well as the methodology and design flow will be explained in Chapter 7.

5.3.1 Instruction Set and Instruction Encoding

The rCSP is an application specific processor targeted towards an effective execution of embedded real-time control, digital filtering and basic signal processing algorithms. Such algorithms consist generally of a piece of code executed endlessly within a loop. The general structure of this code can be divided in three parts: data acquisition, data processing and data outputting. The data processing is the most time consuming section of the algorithm, involving mainly intensive arithmetic processing and data transfers. Therefore the instructions and addressing modes required for coding this section must be highly optimised. As indicated in Section 3.2, this type of arithmetic processing typical to matrix-based signal processing algorithms can be implemented by means of multiplications and additions. It was also seen that the combination of these two operations into a single one (the MAC operation), is the most effective way of expressing such matrix-based real-time equations.

Since the processor is targeted towards the embedded space, size and power consumption must be balanced with performance. The instruction set and the instruction encoding can help to achieve such balance. In first place, a reduced instruction set decreases the logic required for handling and executing the instructions. If only fit-to-purpose instructions are implemented, a significant reduction in size can be obtained. In second place (as seen in Section 3.3.1), the instruction encoding directly affects parameters such as decoding speed and size.

As a result of the previous considerations, the instruction set of the rCSP has been designed as a compact fit-to-purpose ten instruction set (see Table 5.2) with single-cycle execution.

The way in which these instructions cover the general structure of the code executed within the loop is described next. The data processing is covered by six arithmetic instructions (including a MAC instruction) and the data outputting is assured by
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>SUM d,s1,s2</td>
<td>[d]←[s1]+[s2]</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>SUB</td>
<td>SUB d,s1,s2</td>
<td>[d]←[s2]−[s1]</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>MUL</td>
<td>MUL d,a,s1</td>
<td>[d]←[a]·[s1]</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>MAC</td>
<td>MAC d,a,s1,s2</td>
<td>[d]←[a]·[s1]+[s2]</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>MACSAT</td>
<td>MACSAT d,a,s1,s2</td>
<td>[d]←SAT([a]·[s1]+[s2])</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>SGINV</td>
<td>SGINV d</td>
<td>[d]←−[d]</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>LD</td>
<td>LD s1,s2</td>
<td>[s1]←[s2]</td>
<td>Register Transfer</td>
</tr>
<tr>
<td>WRITE</td>
<td>WRITE i,j,s1,s2</td>
<td>Output i ←[s1]</td>
<td>Data Transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRDGT</td>
<td>WRDGT i,s1</td>
<td>Digital Out i ←[s1]</td>
<td>Data Transfer</td>
</tr>
<tr>
<td>WRPWM</td>
<td>WRPWM i,s1</td>
<td>Analogue Out i ←[s1]</td>
<td>Data Transfer</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP n</td>
<td>No Operation (n cycles)</td>
<td>Control</td>
</tr>
</tbody>
</table>

Table 5.2: rCSP Instruction Set, meaning and usage

three different write instructions. However, there is no explicit instruction for reading the input data from an external device. The reason is that both the input buffer and the sampling units are addressable in the same way as the rest of the memory-based modules, having their own memory address space and therefore being accessible directly as an instruction operand. The benefit is threefold: First, the instruction set is smaller, and so is the size of the design and its complexity. Second, the loop overhead is reduced by one instruction per input read. Third, memory space is saved since there is no need to copy the data stored in the input module to the main variables' memory.

It can be seen that there is no unconditional jump instruction available to perform the looping. The reason is that the rCSP has an internal register that is preloaded at booting time with the number of instructions involved in the loop. In this way, a dedicated part of the design is in charge of calculating the address of the next instruction to be fetched and setting the PC to zero when the last instruction of the algorithm is reached. This provides a zero overhead looping mechanism and decreases the instruction set size.

The rCSP uses fixed instruction encoding for fast and easy decoding (see Figure
5.1). All instructions have the same length. The number of operands per instruction varies from zero to four, but their size and positions are constant. The operands (in most cases) must be seen as pointers. They contain the address of the variable or coefficient they refer to. These operands can be identified in Table 5.2 by the notation [operand] and should be read as “the value stored in the address op”, or in C language notation as “*operand”. The write instructions however, are represented in a slightly different way. They are in charge of transferring data from a data memory bank to a number of (analogue and/or digital) output buffers (see Section 5.4.5). Figure 5.2 shows the encoding and meaning of the opcodes and operands for the different rCSP instructions.

![Figure 5.1: rCSP General Instruction Encoding. OPC stands for opcode and Op stands for operand](image)

The instruction WRITE \( i,j,s_1, s_2 \) transfers the values stored in addresses \( s_1 \) and \( s_2 \), to the \( i^{th} \) and \( j^{th} \) digital and analogue output buffers respectively. To transfer data only to digital output buffers the instruction WRDGT \( i,j,s_1, s_2 \) is used. In the same way if only data transfers towards analogue output buffers are required the WRPWM \( i,j,s_1, s_2 \) instruction should be used.

![Figure 5.2: rCSP Instruction Set Encoding](image)

The NOP (Not-Operation) instruction produces a number of idle cycles (see Figure...
5.2 for the encoding) and flushes the pipeline. Therefore, the maximum number of idle (or wait) cycles depends on the rCSP configuration, which determines the instruction word length. In the case depicted in Figure 5.2, the limit of idle cycles is $2^{27} - 1$. This instruction can be used for synchronising the algorithms execution in case of multi-rCSP designs or to accommodate a particular rCSP's external clock oscillator.

5.3.2 Storage Type

As stated in Chapter 3, the internal storage type is one of the most basic differentiations among different processor architectures. The rCSP implements a modified version of the Register-Register architecture class. The traditional Register-Register architecture can be found in many RISC processors and DSPs (e.g. Alpha, Trimedia TM5200, ARM, MIPS, PowerPC, SPARC). The difference with all these implementations is that rCSP's register structure is generated and managed by the compiler and it doesn't change during the algorithm's execution. Therefore, there is no need for LOAD/STORE operations and the data can be accessed by its static address. This is possible due to the predictability in the algorithm's code execution and to the processor's targeted design.

This storage scheme has some important advantages:

- It doesn't require data address calculations, avoiding wasting extra registers and saving clock cycles.
- It doesn't require external memory accesses.
- It doesn't require LOAD/STORE operations to fetch the data, speeding up the algorithm execution.
- It produces clear and compact code.

5.3.3 Memory Addressing

Support for different memory addressing modes come at the cost of design complexity and therefore size and power consumption. Properly choosing the supported address-
ing modes will result in a smaller, faster and less power consuming processor.

In the design of the rCSP, the computational requirements involving control law execution and signal processing were analysed, the memory structure redesign to supply the AU with the required bandwidth and the addressing modes consequently tailored. As a consequence of this analysis, only one addressing mode was implemented: Register.

Table 5.3: Addressing modes supported by rCSP.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>MAC y,a1,x1,x2</td>
<td>REG[y] ← REG[a1] + REG[x1] + REG[x2]</td>
</tr>
</tbody>
</table>

A wide variety of addressing modes can be found in DSP architectures, although as DSP programmers migrate towards larger programs they become more attracted to compilers [22] and as it was previously exposed, compilers are not yet able to map the particularities of certain algorithms into most of these specialised addressing modes. On the contrary, the rCSP implements only one but fit-to-purpose addressing mode.

rCSP’s Internal Data Memory Map

The Internal memory map refers to the data addresses used by the compiler to place the different variables, inputs, targets and coefficients into their particular memory modules. Since there is no jump instruction, the code address is not relevant for the programmer. The rCSP’s internal memory map is static after configuration.

The processor’s modified Harvard architecture not only separates data and code address busses, but also splits the data between different modules, each module containing data of the same type. In that way, different data types can be accessed simultaneously while avoiding the waste of memory that would mean storing them all together on the same memory bank. Table 5.4 shows the internal memory map of the processor. Two different space addressed are provided. The first one is used for addressing static data (i.e. coefficients) while the second one is used for addressing variables, inputs and targets. From a control algorithm point of view, one address space is used for the coefficients’ matrices (A,B,F,G) while the other deals with the
state variables, inputs and references.

As through all the chapter, the following values are related to the processor's standard configuration.

<table>
<thead>
<tr>
<th>Internal address</th>
<th>Variables</th>
<th>Internal address</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 - 15</td>
<td>I-MOD</td>
<td>00 - 127</td>
<td>CFF-MOD</td>
</tr>
<tr>
<td>16 - 31</td>
<td>TGT-MOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 - 127</td>
<td>STV-MOD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: rCSP’s internal data memory map.

5.3.4 Type and Size of Operands

The type of the operands is known by the instruction handler, immediately after decoding the instruction’s opcode. They are tailored for the particular requirements of real-time digital signal processing control algorithms, and therefore, in contrast to desktop and general DSP architectures, they don’t need to be restricted to the typical set of operands (e.g. character, integer, single-precision floating point, etc.). Their sizes are also neither standard nor fixed before compilation. In fact, the size of the different operands can be manually chosen during the rCSP’s configuration process.

Usually, coefficients can have wide dynamic range and therefore they are more suitably represented in floating-point format. The use of the delta operator reduces the coefficients’ sensitivity [12], allowing the use of shorter word-lengths in the coefficients’ representation. Figure 5.3(a) shows the format of the rCSP’s coefficients. The exponent (which is represented as an unsigned integer) is biased to be in the range of [+6, −25]. This reduces the range of gains greater than unity that can be represented, concentrating the maximum floating-point resolution around zero (see Chapter 4) where real-time controllers and filters based on the δ-operator tend to placed most of their coefficients. The mantissa (represented in two’s complement) is also normalised to be in the range [−1, 1). Thus, the value of the coefficients can be calculated as:

$$Cff = \frac{mantissa}{2^5}.2^{(exponent-25)}$$

(5.1)
A more compact expression is:

\[ Cff = \text{mantissa} \cdot 2^{(\text{exponent}-30)} \]  

(5.2)

The dynamic range of the state variables of a controller or filter can be reduced by the scaling properties of the modified canonic delta form (see Section 4.2.2). This allows the use of fixed-point format representation for the state variables. Figure 5.3(b) shows how these variables are internally represented.

![Digital representation of coefficients and variables.](image)

Figure 5.3: Digital representation of coefficients and variables. \( \text{Ovf} \) represents the overflow bits in the fixed-point representation. \( \text{exp} \) represents the exponent of the floating-point representation.

The rCSP implements three different types of operands (see Table 5.5). Each type is stored in a separate register bank of tailored word length, thereby improving the data access and avoiding memory storage overheads.

<table>
<thead>
<tr>
<th>Operand name</th>
<th>Type</th>
<th>Size</th>
<th>Resolution</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Variable</td>
<td>Fixed Point</td>
<td>27-bit</td>
<td>(2^{-12})</td>
<td>([-2^{11}, 2^{11} - 1])</td>
</tr>
<tr>
<td>Coefficient</td>
<td>Floating Point</td>
<td>11-bit</td>
<td>(\approx 3%^\ast)</td>
<td>([-64, 62])</td>
</tr>
<tr>
<td>I/O</td>
<td>Fixed Point</td>
<td>12-bit</td>
<td>Integer</td>
<td>([-2^{11}, 2^{11} - 1])</td>
</tr>
</tbody>
</table>

Table 5.5: Operands types supported by the rCSP.

Section 4.3 provided a detailed explanation about the floating-point formulation used to represent the rCSP's coefficients and their resolution. Due to the uneven distribution of the floating-point numbers, the resolution (\(\ast\)) presented in Table 5.5 represents the worse case scenario. The three guard bits introduced in the state variable type extends the range within these variables can operate to \([-2^{14}, 2^{14} - 1]\). The input signals can be brought into the rCSP in two different ways, namely as a digital signal
or as a signal to be sampled. In both cases, the result is a 12-bit signal \( I_n[11..0] \) as shown in Figure 5.4. Once an input signal is accessed during the algorithm execution, its value is mapped into the structure shown in Figure 5.3(b) to be used as a variable \( V[26..0] \). The mapping is done as follows:

1. \( V[11..0] \leftarrow 0 \) (fractional part equals zero).
2. \( V[23..12] \leftarrow I_n[11..0] \) (integer part transfer).

As explained in Section 3.2.2, overflow bits are a necessity in most digital signal processing applications whose variables are represented in fixed-point format. Due to the two's complement arithmetic format of the variables, the sign is extended to the overflow bits. The fractional (or underflow) bits of the variables, allows for a precision of \( 2^{-n} \) where \( n \) is the number of fractional bits provided (12 in the standard configuration of the rCSP).

![Figure 5.4: Digital representation of the input values](image)

### 5.3.5 Operations in the Instruction Set

Table 5.6 categorises the different operators supported by most instruction set architectures, indicating which ones are supported by the rCSP architecture. As can be seen, only a subset of them are implemented, for the reasons explained in the previous sections.

As Section 3.4 presented, many signal processing applications running on standard processors can benefit by the use of SIMD operations on a partitioned ALU. Due to the targeted architecture of the rCSP, the type and size of the operands are optimised so
Table 5.6: Categories of instruction operators

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Integer Arithmetic</td>
<td>SUM, SUB</td>
</tr>
<tr>
<td>✓ Floating Point</td>
<td>MAC, MUL</td>
</tr>
<tr>
<td>✓ Saturated Arithmetic</td>
<td>SUM, SUB, MAC, MUL</td>
</tr>
<tr>
<td>✓ Single Cycle MAC</td>
<td>MAC</td>
</tr>
<tr>
<td>✓ Data transfer</td>
<td>WRITE, WRDGT, WRPWM</td>
</tr>
<tr>
<td>≈ Control</td>
<td>Only NOP</td>
</tr>
<tr>
<td>X System</td>
<td># (Operating System)</td>
</tr>
<tr>
<td>X Decimal</td>
<td>No need for it</td>
</tr>
<tr>
<td>X String</td>
<td>Operand type not implemented</td>
</tr>
<tr>
<td>X Graphics</td>
<td>Operand type not implemented</td>
</tr>
</tbody>
</table>

The AU is working at full size throughout the whole algorithm execution and therefore no SIMD instructions can be executed with a single arithmetic unit. This is no drawback, it simply means that the AU is well sized and fit-to-purpose. On the other hand, the replication of execution units (typical to SIMD and multi-issue architectures) will imply the replication of the memory banks, leading to an implementation almost as expensive (in logic gates) as two rCSP processors placed in parallel, yet offering less flexibility.

Section 5.7 will show an alternative to SIMD and VLIW, based on multi-rCSP parallel architectures.

5.4 Implementation

The processor's implementation is optimised towards system integration, fit-to-purpose arithmetic execution and embedded real-time control, filtering and signal processing operations.

Figure 5.5 shows a top-level view of the processor's architecture, and Figure 5.6 provides an external view where its input/output signals are grouped by functionality.
Chapter 5: rCSP Architecture

The main functional groups of the processor's internal architecture are listed and briefly described next.

- **Memory storage and management system.** It comprises all the storage units for instructions, data memory and internal registers, with their particular control logic and their management system (MMU). The Coefficient Module (CFF-MOD) stores and manages the algorithm's coefficients, which are periodically updated without interfering with the current algorithm execution. The State Variables Module (STV-MOD) stores the state variables, outputs and all intermediate variables used in the algorithm. It is designed to be externally monitored at any moment in parallel with the algorithm execution. Such algorithm is stored in the Control Law Module (CL-MOD) which also generates and manages the programme counter (PC) logic. The Input Module (I-MOD) stores and manages the digital input values and the analogue sampled ones. It includes successive approximation register (SAR) units and external address generation unit fully integrated with the algorithm's execution, providing a minimal logic solution for interfacing analogue signals. This module can be externally accessed as a memory mapped unit, allowing other devices such (e.g. an embedded CPU) to
access the most recently updated sensors' signals. The **Target Module** (TGT-MOD) stores and manages the control and filter targets, set up for a particular moment or control scenario. The **Internal Register File** (INT-REGS) stores the main parameters of the rCSP as well as some real-time execution information that can be accessed from outside for monitoring and debugging purposes.

- **Input/Output Units.** Two types of converter units are in charge of connecting the processor with the analogue world, namely SAR units and **pulse width modulation** (PWM) units. Digital input and output ports are also available. The number and size of all these units and ports are adjustable to the particular requirements of each application. Section 5.4.5 will describe in detail these units and their associated modules.

- **AMBA interfacing.** The **Advanced Microcontroller Bus Architecture** (AMBA) specification defines an on-chip communications standard for designing high-performance embedded microcontrollers [4]. The **Advanced Peripheral Bus** (APB) is part of the AMBA hierarchy of busses and is optimised for minimal power consumption and reduced interface complexity. The **Advanced High-performance Bus** (AHB) is a new generation of AMBA bus which is intended to address the requirements of high-performance synthesisable designs. It sits above the APB and implements the features required for high-performance, high clock frequency systems. The integration of an AMBA Slave Interface into the rCSP architecture opens a standard communication interface with other processors and external units. Through this interface, the rCSP is commanded, programmed and debugged, the control coefficients adapted and the state variables monitored. An APB slave interface is integrated in the processor's core. The reason is threefold: its design simplicity, its acceptance as one of the industry standards and its perfect suitability to the rCSP's requirements. An AHB adapter is also provided to extend the processor's connectivity.

- **Five stage hybrid unit.** This unit controls the dataflow across the processor, and executes the required arithmetic operations. It comprises the following modules. The **Instruction Handler** (IH) which is in charge of receiving and decoding one instruction per clock cycle, provided the processor is not in stand-by mode.
Chapter 5: rCSP Architecture

The Interlocking module (IL-MOD) whose mission is to bring the processor into the right state by executing the high-level commands sent through the AMBA Interface. And finally, the Arithmetic Unit (AU) which is optimised for executing MAC operations in one clock cycle. Its architecture is not standard, but tailored for performance and space in embedded real-time applications. In order to do so, it uses hybrid arithmetic, where the coefficients are represented in floating-point format and the internal variables in fixed-point format.

In order to be able to properly transfer modulated outputs to analogue actuators the implementation is divided in two different clock domains. The first one (core clock) runs at the maximum speed allowed by the design’s slowest path, and is directly related to the maximum sampling frequency achievable by the system. The second one (pwm core), runs at the maximum speed allowed by PWM unit and the device technology. In this way, the modulated outputs can be updated at much higher rates, therefore keeping up with the sampling frequencies. These clock domains will be described in more detail in Section 5.4.5.

Figure 5.6: rCSP’s pinning overview
5.4.1 Memory System

The rCSP implements a modified Harvard (bus) architecture where the data memory is distributed in separate blocks (a structure more suitable for control and signal processing applications) that can be accessed concurrently.

All the data are stored on-chip in different modules, depending on their types. Their location is determined during compilation and doesn’t change once the processor is running. Therefore, the data can be seen as stored in separated register banks, with independent access to the AU (see Figure 5.7).

![Figure 5.7: rCSP Register-Register storage architecture](image)

There are four separate register banks. The reasons are threefold: First, the arithmetic unit (AU) can process up to four operands at a time. Second, allocating different data types in different register blocks allows the word length of each block to be adjusted avoiding a waste of storage space. Third, this separation of data makes decoding easier and faster, and prevents undesired memory access overlapping. Since the execution of the program is fully predictable, and the memory is on-chip SRAM, there is no need for including any level of cache.

Most of the data modules need to be directly accessed from outside of the processor. This allows, for instance, the state variables and the sampled sensor inputs to be monitored in real-time, the coefficients to be adapted or the control law to be reprogrammed. In order to achieve this without interfering with the algorithm execution, direct memory access (DMA) is available to most of the processor's memory-based modules through its AMBA interface.

Two different memory models are used throughout the whole design:

**Single port SRAM block.** For modules that don’t require multiple read access on the same cycle (CL-MOD and CFF-MOD).

**Dual Port SRAM block with read/write protection.** This architecture allows a read from two memory locations and a write in one memory location in the same clock cycle. In the case that the same address is used to read and write, the data to be written is stored, registered and sent to the output in the next cycle, so no performance penalty nor data corruption occurs. The implementation is shown in Figure 5.8. This memory model is especially suitable for control and signal processing where MAC operations are extensively used.

![Figure 5.8: Dual port memory's internal design](image)

In summary, the memory system has the following characteristics:

- Implemented using on-chip SRAM embedded memory. This provides a single cycle data access and a low power memory storage.
- Separate Memory Banks for different data types.
- Based on Single and Dual-Port modules.
- Read/Write operation allowed on the same clock cycle on the same address.
- DMA available for most of the memory-based modules.
Memory Mapping Unit (MMU)

Once the instruction’s opcode is identified by the Instruction Handler (IH), a set of signals is prepared to drive the instruction’s execution. The first step is to generate the proper read and write addresses to be sent to the different memory modules. Table 5.4 showed the address spaces used by the rCSP’s cross-compiler for managing coefficients, variables, inputs and targets. The addresses of these elements are encoded in the different operands of an instruction.

The task of the rCSP’s MMU is to translate the virtual addresses referred in the operands into the actual address of a particular module where the data is stored. Although this conversion could be done locally on each module, the implementation of a MMU provides a more efficient and centralised solution. In that way for example, if new modules were to be added to the processor’s core, a new entrance in the MMU table would be sufficient to make such modules operational. Figure 5.9 depicts the MMU functional unit in relation with other modules.

![Figure 5.9: rCSP dataflow with MMU](image)

Adaptation and the CFF-MOD

In many applications it is required to be able to adapt the algorithm’s coefficients in real-time depending on certain conditions. The rCSP provides an easy way to externally implement adaptation by means of high-level commands (see Section 5.5 and Chapter 6) that can be received through its AMBA interface.
When a set of new coefficients is ready, the rCSP is informed through its Interlocking Module (IL-MOD). While the coefficients start arriving it is important that the current algorithm's execution is not disrupted. In order to do so, a set of two memory banks are provided. In that way, one of them can hold the currently used coefficients while the other one can load the new ones. Then, after a synchronisation signal arrives (i.e. the PC has reached the last instruction of the algorithm) the CFF-MOD output will be swapped from the current active memory bank to the one that holds the new set of coefficients. The management of the coefficient banks (swapping, writing, enabling, etc.) is done by a state machine embedded in the CFF-MOD. Figure 5.10 shows the internal architecture of the CFF-MOD.

![Figure 5.10: CFF-MOD internal architecture](image)

### 5.4.2 Arithmetic Unit

Due to the nature of the control and digital signal processing algorithms, the arithmetic unit is optimised towards fast single-cycle execution of MAC operations. It accepts three inputs (one coefficient and two state variables) and generates one output (see Figure 5.11).

In order to get a good balance between speed, accuracy, power consumption and size a hybrid approach has been used (see Section 5.3.4), where the coefficients enter
the AU in floating point format, while the state variables enter in fixed-point format. By introducing a pipeline into the AU unit (which is the system's critical path) the speed increased almost by a factor of 2, but an instruction dependency issue arises as a side effect. In order to solve that issue, two Forward-Path levels have been implemented (see Figure 5.12), allowing a stall-free execution of linear control and filtering algorithms. The following section describes in detail the different pipeline hazards that can arise during the execution of an algorithm and how they are solved.

5.4.3 Pipelining and Datapath

As mentioned previously, the rCSP's architecture is based on a modified Harvard architecture (separated data and address bus). The processor has five stages of pipelining, allowing a fast execution and an effective use of the processor resources. Table 5.7 de-
Chapter 5: rCSP Architecture

cribes the different stages of the processor's pipeline during the execution of different instructions. The different pipeline stages correspond with the following operations:

1. **Instruction Fetch.** The Program Counter (PC) requests the next instruction to be executed to the Control Law Module (CL-MOD) where the programme is stored.

2. **Instruction Decoding.** The new instruction released from the CL-MOD arrives into the Instruction Handler (IH). Then, the instruction's opcode is decoded and the control signals are generated. The Memory Management Unit (MMU) receives relevant information from the IH and generates the proper address signals for the different address busses. At the end of this stage, all the addresses and control signals are stable on their busses.

3. **Memory Read and first MAC stage (MUL).** Data is retrieved from the different memory units and reaches the Arithmetic Unit (AU). Three operands (one coefficient and two state variables) are the numerical input to the AU. The control signals generated by the decoding of the instruction in stage two define the operation to be executed. In case of instruction dependencies, control signals will lead the proper data to the AU through a series of forward paths. At the end of this stage, the multiplication between the coefficient's mantissa and the first state variable is executed and its result is stable.

4. **Second MAC stage (Shifting and SUM).** Data is ready. The result from the multiplication in the previous stage is shifted by the coefficient's exponent and its result added to the second state variable. At the end of this stage, the arithmetic operation's result is stable.

5. **Memory Write.** If the executed instruction was arithmetic, its result is written into the State Variable Module (STV-MOD).

**Pipeline Hazards**

Section 3.5.1 showed how pipelining allows the execution speed of a processor to be increased by overlapping the execution of several instructions. This powerful technique
nevertheless can cause problems when for example dependencies exist between the data processed by some of the overlapping instructions. These situations are solved in some cases directly by the hardware, in other cases by the compiler and ultimately some times by the programmer with careful coding. In the case of the rCSP processor, its hardware is in charge of handling such issues.

As can be seen, due to the Harvard architecture of the processor it is possible to fetch a new instruction while performing simultaneously a data memory access. However, a closer observation of Table 5.7 shows that, at the end of the 5-cycle execution of an arithmetic instruction, there is a potential hazard in the data module, in case instruction \( i \) tried to write into the same address that instruction \( i+2 \) reads from. This case is solved in hardware, with a memory design that allows two reads and one write to be performed simultaneously, even to the same address.

![Figure 5.13: Pipeline hazards due to instruction dependency. IM stands for Instruction Memory while DM stands for Data Memory.](image-url)
A pipeline stage has been placed in the AU, which unsurprisingly determines the minimum clock cycle length of the design. Although this technique successfully increases the design speed by almost a factor of two, it also introduces two pipeline hazards due to instruction dependencies (see Figure 5.13). The first hazard happens when an instruction (e.g. instruction $i+2$), tries to read from the memory the result of an instruction executed two cycles before (e.g. instruction $i$). The problem is that instruction $i$ is at stage 5 of execution (i.e. writing the result into the memory), so the value is not ready to be read. This case is solved in hardware by the introduction of an external forward path (see Figure 5.12) that forwards the result obtained at stage 4 (SHIFT-SUM) of instruction $i$ into the stage 3 (READ-MUL) of instruction $i+2$. The second hazard occurs when an instruction (e.g. instruction $i+1$) tries to read from the memory an operand that will be used in stage 4 (SHIFT-SUM) and that it is the result of the previously executed instruction (e.g. instruction $i$). Again, the problem is that instruction $i$ is at stage 5 of execution (i.e. writing the result into the memory), so the value is not ready to be read. This case is also solved in hardware by the inclusion of an internal forward path (see Figure 5.12) that forwards the result obtained at stage 4 (SHIFT-SUM) of instruction $i$ into the stage 4 (SHIFT-SUM) of instruction $i+1$. Figure 5.14 shows how the data is forwarded in the pipeline in the previous cases, avoiding the introduction of any idle cycle.

![Figure 5.14: Data forwarding through different pipeline stages. DM stands for Data Memory](image)

There is the possibility of a third pipeline hazard that happens when an instruction
(e.g. \textit{instruction }i+1\textit{)}, tries to read from the memory an operand that will be used in this very stage (\textit{stage 3, SHIFT-SUM}) and that it is the result of the previously executed instruction (e.g. \textit{instruction }i\textit{)}. In this particular case, the hardware detects the hazard, and it has no other alternative but to introduce an idle cycle to allow \textit{instruction }i+1\textit{ to finish its execution. Obviously there is no path available to forward the result of \textit{instruction }i\textit{ into the \textit{stage 3 (READ-MUL)} of \textit{instruction }i+1\textit{, simply because this result is under calculation at this particular cycle.}

Typically this hazard happens when trying to concatenate two multiplications when the result of the first one is used by the second one. This can be represented in a block diagram as a chain of gains (see Figure 5.15).

![Figure 5.15: Chain of gains leading to a pipeline hazard](image)

This pipeline hazard can be avoided in two ways:

1. To substitute the two gains (\textit{GAIN A, GAIN B}) by a single one
   \[
   \text{\textit{GAIN C= GAIN A·GAIN B}}
   \]

2. To use one of them as pre-gain of a block and the other one as post-gain as shown in Figure 5.16.

If more than two gains are concatenated, then a combination of the previous actions may be required. However, as mentioned before, if this hazard is found, the hardware takes care of handling it by including idle cycles in the execution.

\subsection*{5.4.4 Internal Registers}

The rCSP possesses a number of internal registers that are enumerated and described in Table 5.8.
Figure 5.16: Pre and post-gain configuration that avoids the third pipeline hazard

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Max Value</td>
<td>Holds the algorithm's length minus one (control's loop maximum PC value)</td>
</tr>
<tr>
<td>PWM Shift</td>
<td>Enables or disables the self adjusting of the PWM output for negative values</td>
</tr>
<tr>
<td>PC</td>
<td>Holds the current programme counter (PC) value</td>
</tr>
<tr>
<td>Loop Number</td>
<td>Stores the number of control loops executed since the last RESET command (CMD_RST) was sent to the rCSP.</td>
</tr>
<tr>
<td>Status</td>
<td>Summarises important execution parameters.</td>
</tr>
<tr>
<td>Self Test Mode</td>
<td>If enabled, the processor stops outputting values to external actuators and systems and enters in self testing mode.</td>
</tr>
</tbody>
</table>

Table 5.8: rCSP's internal registers' description

The structure of the internal registers, their access type (i.e. [R]ead and/or [W]rite) and their AMBA address are shown in Figure 5.17.

The Status Register summarises important execution parameters that can be externally accessed in order to monitor the processor's status. Table 5.9 describes these parameters presented with the nomenclature used in Figure 5.17.

The Error State is used to provide information in case an error is found (see Table 5.10). These errors cannot happen in normal circumstances where the provided software tools automatically generate the rCSP's code. However, it can be useful for debugging in case of new features are included in the processor's core.
### Figure 5.17: rCSP's Internal registers

<table>
<thead>
<tr>
<th>Bits</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2..0</td>
<td>ST</td>
<td>rCSP's Current State (see Section 5.5)</td>
</tr>
<tr>
<td>4..3</td>
<td>ERR</td>
<td>Error State (see below)</td>
</tr>
<tr>
<td>5</td>
<td>ePC</td>
<td>If 1 the PC is enabled</td>
</tr>
<tr>
<td>6</td>
<td>eIH</td>
<td>If 1 the IH is enabled</td>
</tr>
<tr>
<td>7</td>
<td>eP</td>
<td>If 1 the PWM Shift is enabled</td>
</tr>
<tr>
<td>8</td>
<td>TM</td>
<td>If 1 the Self Test Mode is active</td>
</tr>
</tbody>
</table>

Table 5.9: Status registers description

#### 5.4.5 I/O Modules

The rCSP includes a number of I/O units to send output signals to other digital and analogue components. In its standard configuration, the rCSP integrates four 12-bit Pulse Width Modulation (PWM) units for digital-to-analogue conversion purposes and can concurrently process up to four analogue external signals with the help of the SAR units included in the I-MOD. More analogue signals can be brought into the rCSP processor by the use of 4 inputs 1 output analogue multiplexors (see Figure 5.18).
## Input Module (I-MOD)

The input module (I-MOD) can be configured to work in digital and analogue environments. Very often, control and filtering applications although implemented in digital devices receive analogue input signals, typically in the form of a current or voltage proportional to certain sensor reading. In order to work with these signals, a digital processor must receive them converted into a suitable digital representation. This conversion, can be done by analogue to digital converters (ADCs). They are a little more complicated that other peripherals because additional control is necessary to start the conversion and to indicate that the process is completed [12]. They can be external to the processor or integrated in the same silicon area, although this option can seriously increase the price of the processor since the fabrication of mixed analogue-digital devices is generally much higher than the digital ones. Another drawback of this solution is the lack of flexibility and waste of silicon area and resources, since a fixed number of ADC units are included in the silicon from fabrication.

Two important ideas in the design of the rCSP were reconfigurability, performance and low cost. Therefore, a different approach was taken in order to interact with the analogue world. The decision was to implement all the digital logic required in the conversion process (SAR units, control and address generation logic) on-chip, while relying on a minimum amount of analogue logic external to the processor. The advantage of this design is threefold: First, it reduces to a minimum the external logic required in the system. Second, it increases the design flexibility since the number of SAR units and external signals can be parameterised and the required logic automatically generated, avoiding wasting resources and saving space and power. Third, the conversion process is controlled by the processor hence it is fully integrated with the
With this architecture, the only required external analogue circuitry is one operational amplifier and one digital-to-analogue converter (DAC) per input to the rCSP. In case that many analogue inputs are present, the use of a number of analogue multiplexers (A-MUX) would be sensible. Figure 5.18 provides a functional view of the I-MOD and the external analogue circuitry required for processing $4 \cdot N$ analogue inputs. The inputs are multiplexed in sets of four inputs per A-MUX. DACs are, in general, relatively cheap and simple devices and Successive Approximation Converters (SACs) are most commonly used in control [12] and so this implementation scheme was chosen. More information and other analogue and digital conversion architectures can be found for example in [21] and [41].

![I-MOD architecture implementing SAR converters](image)

Of course, the fact that the rCSP includes SAR converters does not prevent its use with external ADC units. In fact this option, if available for a particular project, could result in an easier implementation, where analogue design problems such as SAR-DAC-Operational Amplifiers interfacing could be avoided.

Many times the processor receives digital information from sensors (digital ones or form a ADC) or other digital stages. Figure 5.19 shows a functional diagram of the
I-MOD configured for receiving digital inputs. As shown in the figure the inputs can be (if chosen) registered.

![Diagram of I-MOD architecture optimised for digital inputs]

Figure 5.19: I-MOD architecture optimised for digital inputs

**Output Module**

The output module is in charge of sending the value of selected variables or inputs outside of the processor to other digital and analogue stages. If the output is a variable in fixed-point format, its value is rounded to the nearest integer. A basic scheme of a single output block is depicted in Figure 5.20. Each block has the possibility to generate a digital output (i.e. to be transfer to another digital stage or to feed the input of a DAC) and/or to generate a single bit pulse width modulated (PWM) output signal (the pulse width of which is made proportional to the output variable). Taking this signal through a low pass filter generates an analogue output. However, in applications where the actuator or the plant includes sufficient inductance (e.g. the coils of a motor or a magnetic levitation vehicle), no extra low pass filter is required.

PWM generated signals require a modulation frequency much higher than the sampling one. Thus, the same clock that drives the processor’s core cannot drive the PWM units and hence the design needs to be divided into two clock domains. The first one (*core clock*) runs at the maximum speed allowed by the design’s slowest path, and is directly related to the maximum sampling frequency achievable by the system. The second one (*pwm clock*), runs at the maximum speed allowed by PWM unit and
the device technology. In this way, the modulated outputs can be updated at much higher rates, therefore keeping up with the sampling frequencies.

The PWM clock frequency depends on the word length of the digital value to be converted. In terms of the sampling frequency \( f_{\text{sampling}} \), the PWM clock frequency of an \( N \)-bit digital signal is given by Equation (5.3).

\[
f_{\text{PWM}} = 2^N \cdot f_{\text{sampling}}
\]  

The same frequency can be calculated in terms of the rCSP clock frequency, by defining first a coefficient \( K \) that expresses how many times the PWM clock must be faster than the rCSP core clock in order to be able to output at the same frequency as the sensors are sampled (ideal case). For an algorithm with \( m \) instructions, \( K \) is equal to

\[
K = \frac{2^N}{m}
\]  

Finally, Equation (5.5) presents the PWM clock frequency in terms of the rCSP clock frequency.

\[
f_{\text{PWM}} = K \cdot f_{\text{core}}
\]  

Due to its dedicated clock domain the PWM output ratio is not limited by the architecture, but by the maximum clock speed allowed by the selected implementation technology (FPGA or ASIC). In case the technology imposes lower clock speeds, the actual PWM core frequency obtained would be:

\[
f_{\text{PWM,real}} = K' \cdot f_{\text{core}}
\]

where \( K' \leq K \). In that case, the number of samples that contributes to a single PWM output is given in Equation 5.7 and graphically depicted in Figure 5.21.
Chapter 5: rCSP Architecture

\[ \phi = \frac{f_{PWM \, ideal}}{f_{PWM \, real}} \quad \text{where } \phi > 1 \]  

(5.7)

Figure 5.21: Samples contribution to the PWM output, for a ratio of 3

However, in many real applications and especially in real-time systems where inputs are oversampled, the difference between consecutive sampled values tends to be very small. In many cases the outputs generated for each sample differ only in some fractional parts that are anyway either trimmed or rounded before being sent to the PWM unit. Therefore, the results obtained by Equation 5.6 are often sufficient provided that \( \phi \) is not too big. Figure 5.22 shows the PWM functional diagram with the clock domains.

5.5 AMBA External Interface and IL-MOD

The rCSP can interface to external devices via its built-in AMBA (APB or AHB) slave interface, exposing some of its modules as memory mapped components that
can be directly accessed by means of AMBA READ/WRITE operations. However, in order to perform a write operation, first the Interlocking Module (IL-MOD) must be notified of the targeted module and the type of intended operation. The high-level commands used for interacting with the IL-MOD are summarised in Table 5.11. Figure 5.23 shows the rCSP's internal states and the transitions triggered by these commands. The state's management is performed by a state machine integrated into the IL-MOD.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD_RST</td>
<td>RESETs the rCSP state machines</td>
</tr>
<tr>
<td>CMD_NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>CMD_PRG</td>
<td>Commands the rCSP into CL-MOD programming mode</td>
</tr>
<tr>
<td>CMD_EPR</td>
<td>Notify the end of the CL-MOD programming mode</td>
</tr>
<tr>
<td>CMD_ADP</td>
<td>Commands the rCSP into CFF-MOD programming mode</td>
</tr>
<tr>
<td>CMD_EAD</td>
<td>Notify the end of the CFF-MOD programming mode</td>
</tr>
<tr>
<td>CMD_RUN</td>
<td>Commands the rCSP into algorithm execution mode</td>
</tr>
<tr>
<td>CMD_STP</td>
<td>STOPs the algorithm execution mode</td>
</tr>
</tbody>
</table>

Table 5.11: High-level commands accepted by the IL-MOD

In order to interface an external device with the rCSP a valid AMBA address must be generated following the structure shown in Figure 5.24. Table 5.12 describes the different sections of a valid address.

<table>
<thead>
<tr>
<th>Address Bits</th>
<th>Functionality Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[14]</td>
<td>Selection bit used by the embedded CPU (see Section 6.3.1)</td>
</tr>
<tr>
<td>[13 .. 10]</td>
<td>Decoding. These bits select the targeted module</td>
</tr>
<tr>
<td>[9 .. 2]</td>
<td>Targeted Module Internal Address (256 positions)</td>
</tr>
<tr>
<td>[1 .. 0]</td>
<td>Not Used (AMBA's 4-byte alignment)</td>
</tr>
</tbody>
</table>

Table 5.12: AMBA address partitioning

Table 5.13 shows the list of AMBA base addresses ($AMBA_{BA}$) that point to the first element of each memory mapped rCSP module. In order to access the $n$-th element of a particular module (if such element exists), the AMBA effective address
Figure 5.23: rCSP's internal states and transitions

Figure 5.24: AMBA address word structure

\[(\text{AMBA}_{EA})\] must be calculated as:

\[\text{AMBA}_{EA} = \text{AMBA}_{BA} + n \cdot 4 \]  

(5.8)

If an illegal (nonexisting) address is accessed, the rCSP will return:
\[\text{AMBA}_{RDATA} = \underbrace{1111 \ldots 11111} \]

The standard configuration of the rCSP reserves nine extra base addresses that can be used to include up to nine new memory mapped modules into its core, hence extending its functionality and making it easily upgradable. These auxiliary modules are referred in Table 5.13 as \[\text{AUX}_n - \text{MOD}\].

As can be seen, only the four least significant bytes are used to generate a valid
Table 5.13: List of memory mapped modules and their corresponding AMBA base addresses

<table>
<thead>
<tr>
<th>AMBA Base Address</th>
<th>Decoding Address</th>
<th>Module</th>
<th>Type of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0b0000</td>
<td>INT-REGS</td>
<td>R/W</td>
</tr>
<tr>
<td>0x00000040</td>
<td>0b0001</td>
<td>TGT-MOD</td>
<td>W</td>
</tr>
<tr>
<td>0x00000080</td>
<td>0b0010</td>
<td>IL-MOD</td>
<td>W</td>
</tr>
<tr>
<td>0x000000C0</td>
<td>0b0011</td>
<td>STV-MOD</td>
<td>R</td>
</tr>
<tr>
<td>0x00001000</td>
<td>0b0100</td>
<td>CFF-MOF</td>
<td>W</td>
</tr>
<tr>
<td>0x00001400</td>
<td>0b0101</td>
<td>CL-MOD</td>
<td>W</td>
</tr>
<tr>
<td>0x00001800</td>
<td>0b0110</td>
<td>I-MOD</td>
<td>R</td>
</tr>
<tr>
<td>0x00001C00</td>
<td>0b0111</td>
<td>AUX1-MOD</td>
<td>R/W</td>
</tr>
<tr>
<td>0x00002000</td>
<td>0b1000</td>
<td>AUX2-MOD</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00003C00</td>
<td>0b1111</td>
<td>AUX9-MOD</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Bit 14 is the first bit that is not used in the addressing of the different modules. Therefore, it is the first bit that can be used by the commanding device (usually an embedded CPU) to identify (i.e. $AMB_AEA[14]=1$) the rCSP processor among the rest of the peripherals that are interfaced. However, the peripheral mapping is entirely dependent on the commanding device's configuration and therefore any other bit can be used provided it is within the range [31..14]. Then, the final address can be calculated as:

$$AMB_AAddr = AMB_AEA + 2^k$$  \(5.9\)

where $k$ represents the chosen selection bit. Therefore the base address of the rCSP (i.e. the base address of its first memory mapped module) should be a multiple of 0x4000.

The following example summarises what has been presented in this section in terms of AMBA external interfacing. In this particular example no selection bit is used.

% Read an internal variable at address 0xFF

```plaintext
AMBA_ADDR=0x000001008 % STV-MOD selected - Internal Address: 0x02
AMBA_WDATA=(not used) % Read Operation
AMBA_RDATA=stv[0x02] % stv[0x02] is returned in the next cycle.
```
% Commanding the rCSP into Adaptation Mode
AMBA_ADDR=0x00000800 % IL-MOD selected
AMBA_WDATA=CMD_ADP % Command = Prepare to receive a new set of coeffs.

% Update a coefficient at address 0xFF
AMBA_ADDR=0x000013FC % CFF-MOD selected - Internal Address: 0xff
AMBA_WDATA=0x00000011 % coeff[0xff]=0x11

5.5.1 Software considerations

All the software routines required for interfacing the rCSP with an embedded CPU (drivers) are provided as an ANSI-C library and therefore the user doesn’t need to worry about any low level programming issues or AMBA address calculations. Chapter 7 gives a detailed description of the rCSP Development Suite and Appendix C summarises the ANSI-C device-driver libraries provided for interfacing the rCSP with an embedded CPU on a system-on-chip solution. In case the rCSP is used as stand alone controller, software tools generate a high-level boot code that is automatically post-processed to produce a synthesisable VHDL boot module.

5.6 Reconfigurability

The processor can be tailored for each particular application, avoiding any waste of resources (especially important in embedded applications), optimising its implementation while reducing power consumption. Speed can be also improved depending mainly on the selected variable and coefficient sizes. Its reconfigurable design makes implementations based on FPGA technologies especially interesting due to the possibility of updating and adapting not only the algorithm but the design parameters without the need for re-engineering the system architecture. The rCSP processor is reconfigurable at VHDL level. The explicit use of hardcoded values to specify the size of the different subsystems’ architectural elements and busses are avoided, favouring more
flexible implementation structures such as custom composite and enumeration types. In that way, the whole processor structure is automatically generated according to the parameters set in the configuration files. Composite types in VHDL consist of arrays and record types. Arrays are groups of elements of the same type (typically used for modelling linear structures such as buses, registers and memories), while record types allow the grouping of elements of different types. Records are especially useful when designing complex IP blocks to group associated signal (according to functionality and direction), obtaining both shorter and more readable port lists, and simplifying the modification of the interface list. An unconstrained array is a type whose range or size is not completely specified when the type is declared, allowing the declaration of multiple subtypes that share a common base type. Composite and enumerated types allow very abstract modelling of hardware [43] and therefore they are widely used throughout the rCSP processor code. Since the size of the different elements are not fixed, the calculation of the proper busses, addresses and word length sizes required to interface the different modules is often not straightforward. In order to simplify future upgrades in the processor’s architecture and to increase the code readability, the components’ interfacing is not explicitly calculated in-situ, but returned by VHDL function calls. These functions are all located on the file rcsp.core.ifaces.vhd, which also defines the interfacing among the different modules that comprises the rCSP core.

Since the interface definitions are all relative, depending on the configuration parameters, a configuration file is required to obtain the final structure. This configuration file is rcsp.core.config.vhd, which as its name indicates defines the main parameters of the rCSP’s core. An extract of this file is provided next:

```vhdl
-- General Implementation parameters
constant DEF_CORE_FPGA_ALTERA : boolean := false;
constant DEF_CORE_FPGA_XILINX : boolean := false;
constant DEF_CORE_FPGA_GENERIC : boolean := true;
constant DEF_CORE_SAR_ENABLE : boolean := true;
constant DEF_CORE_ADJUST_INSTR ALWAYS : boolean := false;
constant DEF_CORE_BOOT_FROM_PROM : boolean := true;

-- CL-HID parameters
constant DEF_CORE_INSTRUCTION_MEMORY_LINES : Integer := 256;
```
Chapter 5: rCSP Architecture

--- Arithmetic Unit parameters
constant DEF_CORE_COEFFICIENT_LINES : Integer := 128;
constant DEF_CORE_STV_LINES : Integer := 128:

--- Coefficient architecture
constant DEF_CORE_COEFFICIENT_MANTISSA_WL : Integer := 6;
constant DEF_CORE_COEFFICIENT_EXPONENT_WL : Integer := 5;

--- State Variables architecture
constant DEF_CORE_STV_INTEGER_WL : Integer := 12;
constant DEF_CORE_STV_FRAC_WL : Integer := 12;
constant DEF_CORE_STV_GUARD_WL : Integer := 3;

--- Input/Output Parameters
constant DEF_CORE_IO_WL : Integer := 12;
constant DEF_CORE_OMOD_BLOCKS : Integer := 4;
constant DEF_CORE_IMOD_BLOCKS : Integer := 4;

On the other hand, the file rcspp_on_chip_config.vhd defines the basic interfaces and parameters that allow the rCSP to be booted from a PROM or to be integrated as a soft IP core in as system-on-chip architecture. In this file, for instance, the AMBA address and configuration parameters of different rCSP modules is defined. This information can be used by other IP cores (e.g. an embedded CPU) to access them in a register-mapped fashion through the rCSP's AMBA slave interface. This file also configures the PROM parameters if a stand-alone implementation is selected. A summary of the rCSP's VHDL model hierarchy is provided in Appendix B.

5.7 Parallel Architectures

Chapter 3 introduced a number of architectures developed to exploit different levels of parallelism. As shown in this chapter, the rCSP is a single-issue architecture that focus on instruction level parallelism (pipelining). It doesn't implement a SIMD architecture for two main reasons. First, its arithmetic unit and operands size are optimised in a way that no smaller data sizes can be obtained to feed a partitioned arithmetic unit. Second the arithmetic unit works at full capacity all the time. Another implementation approach could have lead to the inclusion of a number of datapaths and arithmetic units in a VLIW multi-issue fashion, although due to the sheer performances obtained by the single-issue architecture, this option does not provide any real advantage in most cases, diverting the processor's design from its original fit-to-purpose
goal.

However, as computational and functional requirements increase, parallelism could be an asset and therefore the processor and the rCSP Development Suite have been developed to support the design, simulation and implementation of multiprocessor rCSP designs. Two main applications are presented next.

Algorithm speed up

The idea is based on the parallel execution of a number of rCSP processors that communicate partial results of the algorithm being processed to each other.

Figure 5.25 shows an example where an algorithm has been split into three different sections, each one implemented on a different rCSP processor. In this example, the inputs $A$ and $B$ are analogue signals and the output $C$ is a modulated action to be sent to a particular actuator. Processors one ($P1$) and two ($P2$) are in charge of pre-filtering the input signal $B$, sending the resulting signal to processor number three ($P3$). $P3$ receives three inputs. The first one ($A$) is an analogue signal that is processed while $P1$ and $P2$ are operating on signal $B$. The other two input signals are the digital outputs generated by $P1$ and $P2$. After processing $A$ and receiving the other two digital inputs, $P3$ executes a control law, generating a modulated output ($C$).

Although the most efficient execution is theoretically achieved by dividing the original algorithm in three different parts of equal computational size (for this particular example), in most cases this is not possible or even convenient and different processors will probably finish their execution cycles at different moments in time. Their synchronisation is made by injecting wait statement via the NOP n instruction at the end of the algorithm.

Fault Detection

In environments where fault detection is necessary, a number of rCSP processors (typically 3) could be executing the same algorithm in parallel. Then, an independent logic
Figure 5.25: Multi-rCSP architecture for faster algorithm execution.

block should be able to periodically compare their outputs and decide (vote) whether the system is working fine (the three outputs are identical), a SUE (single upset event) has corrupted the calculation of one processor, so this processor is disregarded (two outputs are identical), or the system is malfunctioning and it should be halted (the three outputs are different).

5.8 Implementation Figures (Standard Configuration)

The implementation of the rCSP has been done targeting two major FPGA vendors (Xilinx and Altera). The results provided in this section are related to the particular technologies listed in Table 5.14. Appendix A presents an overview of the main architectural parameters of these devices. The synthesis tool used was Synplify Pro v.7.3.1 (Built: Jun 2 2003) from Synplicity. The place & route tools used were: Xilinx's ISE (Integrated Software Environment) v.6.1 (for Xilinx devices) and Altera's Quartus II v. 4.1 for the Excalibur ARM FPGA.

rCSP's memory figures

The rCSP processor in its standard configuration requires less than 3 Kbytes of FPGA's memory. Table 5.15 and Figure 5.26 show how this memory is distributed
Chapter 5: rCSP Architecture

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Technology</th>
<th>Part</th>
<th>Package</th>
<th>Speed</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera</td>
<td>Excalibur ARM</td>
<td>EPXA10</td>
<td>FC1020</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Spartan-IIIE</td>
<td>XC2S200E</td>
<td>FG456</td>
<td></td>
<td>-6</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Spartan-IIIE</td>
<td>XC2S600E</td>
<td>FG456</td>
<td></td>
<td>-6</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Virtex-II</td>
<td>XCV1000</td>
<td>BG575</td>
<td></td>
<td>-6</td>
</tr>
</tbody>
</table>

Table 5.14: Targeted FPGAs

among the different memory-based modules of the processor.

<table>
<thead>
<tr>
<th>Module</th>
<th>Words</th>
<th>Bits</th>
<th>Bytes</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFF-MOD</td>
<td>128</td>
<td>2816</td>
<td>352</td>
<td>12.6%</td>
</tr>
<tr>
<td>STV-MOD</td>
<td>128</td>
<td>10368</td>
<td>1296</td>
<td>46.5%</td>
</tr>
<tr>
<td>TGT-MOD</td>
<td>16</td>
<td>384</td>
<td>48</td>
<td>1.7%</td>
</tr>
<tr>
<td>I-MOD</td>
<td>16</td>
<td>576</td>
<td>72</td>
<td>2.6%</td>
</tr>
<tr>
<td>CL-MOD</td>
<td>256</td>
<td>8192</td>
<td>1024</td>
<td>36.6%</td>
</tr>
<tr>
<td>Total</td>
<td>N/A</td>
<td>22336</td>
<td>2792</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 5.15: rCSP's memory distribution

Figure 5.26: Memory distribution among different rCSP modules

How much of the rCSP's memory is used depends on the algorithm implemented, and therefore is device-independent. Table 5.16 presents the memory utilisation figures of a number of algorithms compiled for rCSP. For each algorithm, three different values are provided, namely number of coefficients in the algorithm, number of variables in the algorithm and number of rCSP instructions. The percentages included in this table express the level of utilisation of each resource. The last two algorithms
(i.e. the fly-by-wire controller and the HDD controller) were used in Chapter 7 as demonstrators.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Coefficients</th>
<th>Variables</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Magnet</td>
<td>39 (30.4%)</td>
<td>28 (29.1%)</td>
<td>51 (19.9%)</td>
</tr>
<tr>
<td>Filter (4th order)</td>
<td>10 (7.8%)</td>
<td>6 (6.2%)</td>
<td>14 (5.4%)</td>
</tr>
<tr>
<td>Filter (16th order)</td>
<td>34 (26.5%)</td>
<td>18 (18.7%)</td>
<td>50 (19.5%)</td>
</tr>
<tr>
<td>Filter (46th order)</td>
<td>94 (73.4%)</td>
<td>48 (50%)</td>
<td>140 (54.6%)</td>
</tr>
<tr>
<td>Fly-by-wire controller</td>
<td>16 (12.5%)</td>
<td>16 (12.5%)</td>
<td>20 (7.8%)</td>
</tr>
<tr>
<td>HDD controller</td>
<td>21 (16.4%)</td>
<td>11 (8.6%)</td>
<td>26 (10.1%)</td>
</tr>
</tbody>
</table>

Table 5.16: rCSP's memory utilisation figures for different algorithms

As Table 5.16 shows, the standard configuration of the rCSP is versatile enough to execute a wide variety of algorithms including high-order filters without reaching its resource limit.

**Speed figures**

The sample frequency achievable by the rCSP depends on two main factors, namely the implementation technology and the control algorithm. The post place & route clock frequencies obtained for the targeted FPGA technologies are summarised in Table 5.17.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Part</th>
<th>Core-clock</th>
<th>PWM-clock</th>
<th>Instruction delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excalibur ARM</td>
<td>EPXA10</td>
<td>32 (MHz)</td>
<td>120 (MHz)</td>
<td>31.25</td>
</tr>
<tr>
<td>Spartan IIE</td>
<td>XC2S200E</td>
<td>36.5 (MHz)</td>
<td>120 (MHz)</td>
<td>27.39</td>
</tr>
<tr>
<td>Spartan IIE</td>
<td>XC2S600E</td>
<td>34 (MHz)</td>
<td>120 (MHz)</td>
<td>29.41</td>
</tr>
<tr>
<td>Virtex2</td>
<td>XCV1000</td>
<td>80 (MHz)</td>
<td>266 (MHz)</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Table 5.17: rCSP's clock frequencies

Table 5.18 summarises the loop delays of the test algorithms presented in Table 5.16 based on the maximum clock-frequencies presented in Table 5.17. The inverse of
the loop delay is the maximum sample frequency that can be achieved. These values are summarised in Table 5.19 for each FPGA device.

<table>
<thead>
<tr>
<th>Model</th>
<th>Spartan-II E</th>
<th>Spartan-II E</th>
<th>Virtex-II</th>
<th>Excalibur</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC2S200E</td>
<td>XC2S600E</td>
<td>XCV1000</td>
<td>EPXA10</td>
</tr>
<tr>
<td>Single Magnet</td>
<td>1,397.2 ns</td>
<td>1,500 ns</td>
<td>637.5 ns</td>
<td>1,593.8 ns</td>
</tr>
<tr>
<td>Filter (4th order)</td>
<td>383.5 ns</td>
<td>411.6 ns</td>
<td>175 ns</td>
<td>437.5 ns</td>
</tr>
<tr>
<td>Filter (16th order)</td>
<td>1,369.8 ns</td>
<td>1,470.5 ns</td>
<td>625 ns</td>
<td>1,562.5 ns</td>
</tr>
<tr>
<td>Filter (46th order)</td>
<td>3,835.6 ns</td>
<td>4,117.6 ns</td>
<td>1,750 ns</td>
<td>4,375 ns</td>
</tr>
<tr>
<td>Fly-by-wire controller</td>
<td>547.9 ns</td>
<td>588 ns</td>
<td>250 ns</td>
<td>625 ns</td>
</tr>
<tr>
<td>HDD controller</td>
<td>712.3 ns</td>
<td>764.7 ns</td>
<td>325 ns</td>
<td>812.5 ns</td>
</tr>
</tbody>
</table>

Table 5.18: Loop delay figures

<table>
<thead>
<tr>
<th>Model</th>
<th>Spartan-II E</th>
<th>Spartan-II E</th>
<th>Virtex-II</th>
<th>Excalibur</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC2S200E</td>
<td>XC2S600E</td>
<td>XCV1000</td>
<td>EPXA10</td>
</tr>
<tr>
<td>Single Magnet</td>
<td>715 KHz</td>
<td>666 KHz</td>
<td>1.568 MHz</td>
<td>627 KHz</td>
</tr>
<tr>
<td>Filter (4th order)</td>
<td>2.607 MHz</td>
<td>2.4285 MHz</td>
<td>5.714 MHz</td>
<td>2.285 MHz</td>
</tr>
<tr>
<td>Filter (16th order)</td>
<td>730 KHz</td>
<td>680 KHz</td>
<td>1.600 MHz</td>
<td>640 KHz</td>
</tr>
<tr>
<td>Filter (46th order)</td>
<td>260 KHz</td>
<td>242 KHz</td>
<td>571.4 KHz</td>
<td>228 KHz</td>
</tr>
<tr>
<td>Fly-by-wire controller</td>
<td>1.825 MHz</td>
<td>1.700 MHz</td>
<td>4.000 MHz</td>
<td>1.600 MHz</td>
</tr>
<tr>
<td>HDD controller</td>
<td>1.400 MHz</td>
<td>1.300 MHz</td>
<td>3.076 MHz</td>
<td>1.230 MHz</td>
</tr>
</tbody>
</table>

Table 5.19: Maximum sampling frequency figures

**Complexity figures**

The complexity figures of the rCSP processor are summarised in Table 5.20 for the Excalibur FPGA and in Table 5.21 for Xilinx devices. As Table 5.21 shows, the rCSP in its standard configuration can be successfully implemented in low-cost FPGA families such as Spartan-II E, including its low gate count model XC2S200E.
An RTL view of the rCSP's core generated by the synthesis tool is presented in Figure 5.27.

<table>
<thead>
<tr>
<th>LEs</th>
<th>ESB</th>
<th>Memory bits</th>
<th>PLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,271 (8%)</td>
<td>20 (13%)</td>
<td>22,528 (6%)</td>
<td>1 (25%)</td>
</tr>
</tbody>
</table>

Table 5.20: rCSP's complexity figures for the Excalibur EPXA10 device

<table>
<thead>
<tr>
<th></th>
<th>Spartan-IIE</th>
<th>Spartan-IIE</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC2S200E</td>
<td>XC2S600E</td>
<td>XCV1000</td>
</tr>
</tbody>
</table>

**Logic Utilisation**
- Number of Slice Flip Flops: 1122 (23%), 1130 (8%), 1078 (10%)
- Number of 4 input LUTs: 2515 (53%), 2534 (18%), 2246 (21%)

**Logic Distribution**
- Number of occupied Slices: 1997 (84%), 2013 (29%), 1866 (36%)
- Total 4 input LUTs: 3397 (72%), 3416 (24%), 3123 (30%)
- Used as logic: 2515, 2534, 2246
- Used as a route-thr.: 98, 98, 93
- Used for Dual Port RAMs: 784, 784, 784

<table>
<thead>
<tr>
<th></th>
<th>Spartan-IIE</th>
<th>Spartan-IIE</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC2S200E</td>
<td>XC2S600E</td>
<td>XCV1000</td>
</tr>
</tbody>
</table>

**5.9 Implementation Figures (Various Configurations)**

This section presents the implementation figures for a variety of rCSP's arithmetic unit configurations. The complexity and speed figures have been summarised together
in two tables depending on the Xilinx’s device family used for synthesis and place & route. In that way, the Spartan-IIIE figures are shown in Table 5.22 while the Virtex-II figures are summarised in Table 5.23.

A number of arithmetic unit configurations have been obtained by varying the number of bits used for representing the coefficient’s mantissa. Other parameters could have been modified, but the coefficient’s mantissa is typically the most influential in terms of accuracy, processing performance and design complexity.

<table>
<thead>
<tr>
<th></th>
<th>XC2S200E</th>
<th>XC2S300E</th>
<th>XC2S300E</th>
<th>XC2S600E</th>
<th>XC2S600E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>m=6</td>
<td>m=6</td>
<td>m=11</td>
<td>m=6</td>
<td>m=11</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>core-clock</td>
<td>36.5 MHz</td>
<td>34.4 MHz</td>
<td>29 MHz</td>
<td>34 MHz</td>
<td>31 MHz</td>
</tr>
<tr>
<td>pwm-clock</td>
<td>12 MHz</td>
<td>12 MHz</td>
<td>12 MHz</td>
<td>12 MHz</td>
<td>12 MHz</td>
</tr>
<tr>
<td><strong>Equivalent gate count</strong></td>
<td>274,612</td>
<td>274,614</td>
<td>286,356</td>
<td>274,775</td>
<td>286,375</td>
</tr>
<tr>
<td><strong>Logic Utilisation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>1122 (23%)</td>
<td>1123 (18%)</td>
<td>1081 (17%)</td>
<td>1130 (8%)</td>
<td>1078 (7%)</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>2515 (33%)</td>
<td>2514 (40%)</td>
<td>2682 (43%)</td>
<td>2534 (18%)</td>
<td>2690 (19%)</td>
</tr>
<tr>
<td><strong>Logic Distribution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>1997 (94%)</td>
<td>1998 (65%)</td>
<td>1882 (61%)</td>
<td>2013 (29%)</td>
<td>1897 (27%)</td>
</tr>
<tr>
<td><strong>Total 4 input LUTs</strong></td>
<td>3397 (72%)</td>
<td>3396 (55%)</td>
<td>3223 (82%)</td>
<td>3416 (24%)</td>
<td>3213 (23%)</td>
</tr>
<tr>
<td>... used as logic</td>
<td>2515</td>
<td>2514</td>
<td>2682</td>
<td>2534</td>
<td>2690</td>
</tr>
<tr>
<td>... used as a route-thr.</td>
<td>98</td>
<td>98</td>
<td>109</td>
<td>98</td>
<td>109</td>
</tr>
<tr>
<td>... used for Dual Port RAMs</td>
<td>784</td>
<td>784</td>
<td>452</td>
<td>784</td>
<td>432</td>
</tr>
<tr>
<td><strong>Number of bonded IOBs</strong></td>
<td>192 (67%)</td>
<td>192 (59%)</td>
<td>192 (59%)</td>
<td>192 (59%)</td>
<td>192 (59%)</td>
</tr>
<tr>
<td><strong>Number of Block RAMs</strong></td>
<td>12 (55%)</td>
<td>12 (75%)</td>
<td>14 (87%)</td>
<td>12 (16%)</td>
<td>14 (19%)</td>
</tr>
<tr>
<td><strong>Number of GCLKs</strong></td>
<td>2 (50%)</td>
<td>2 (20%)</td>
<td>2 (50%)</td>
<td>2 (12%)</td>
<td>2 (50%)</td>
</tr>
</tbody>
</table>

Table 5.22: Implementation figures for different mantissa sizes (Spartan-IIIE family)

5.10 Summary

This chapter described in detail the internal architecture, implementation and functionality of the rCSP processor. Its different modules were analysed, and its configuration files reviewed. The instruction set and the execution’s datapath were also presented. The external communication channel was exposed, clarifying the rCSP’s interfacing with an external processing device (e.g. embedded CPU). At the end of the chapter the speed and complexity figures of the processor were presented.
### Table 5.23: Implementation figures for different mantissa sizes (Virtex-II)

<table>
<thead>
<tr>
<th>Various mantissa sizes</th>
<th>( m=6 )</th>
<th>( m=8 )</th>
<th>( m=10 )</th>
<th>( m=11 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>core-clock</td>
<td>80 MHz</td>
<td>68.5 MHz</td>
<td>71.7 MHz</td>
<td>69.2 MHz</td>
</tr>
<tr>
<td>pwm-clock</td>
<td>240 MHz</td>
<td>240 MHz</td>
<td>250 MHz</td>
<td>266 MHz</td>
</tr>
<tr>
<td><strong>Equivalent gate count</strong></td>
<td>557,307</td>
<td>665,864</td>
<td>673,151</td>
<td>743,551</td>
</tr>
<tr>
<td><strong>Logic Utilisation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>1078 (10%)</td>
<td>1082 (10%)</td>
<td>1086 (8%)</td>
<td>1076 (10%)</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>2246 (21%)</td>
<td>2359 (22%)</td>
<td>2126 (20%)</td>
<td>2224 (22%)</td>
</tr>
<tr>
<td><strong>Logic Distribution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>1866 (36%)</td>
<td>1889 (36%)</td>
<td>1844 (36%)</td>
<td>1704 (33%)</td>
</tr>
<tr>
<td>Total 4 input LUTs</td>
<td>3123 (30%)</td>
<td>3204 (31%)</td>
<td>3130 (30%)</td>
<td>2851 (27%)</td>
</tr>
<tr>
<td>... used as logic</td>
<td>2246</td>
<td>2259</td>
<td>2126</td>
<td>2324</td>
</tr>
<tr>
<td>... used as a route-thr.</td>
<td>93</td>
<td>97</td>
<td>92</td>
<td>95</td>
</tr>
<tr>
<td>... used for Dual Port RAMs</td>
<td>784</td>
<td>848</td>
<td>912</td>
<td>432</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>194 (59%)</td>
<td>194 (59%)</td>
<td>194 (59%)</td>
<td>194 (59%)</td>
</tr>
<tr>
<td>Number of Block RAMs</td>
<td>8 (20%)</td>
<td>8 (20%)</td>
<td>8 (20%)</td>
<td>10 (25%)</td>
</tr>
<tr>
<td>Number of MULT18X18s</td>
<td>2 (5%)</td>
<td>2 (5%)</td>
<td>2 (5%)</td>
<td>2 (5%)</td>
</tr>
<tr>
<td>Number of GCLKs</td>
<td>2 (12%)</td>
<td>2 (12%)</td>
<td>2 (12%)</td>
<td>2 (12%)</td>
</tr>
</tbody>
</table>
Figure 5.27: Annotated RTL view of the rCSP's core
Chapter 6

System-on-Chip Solution for Embedded Real-Time Control

6.1 Introduction and Objectives of the Chapter

Only 2% of the world’s microprocessors run on traditional desktops and servers [33]. The rest reside inside something else (thus the adjective “embedded”) whether that’s a digital camera or an MP3 player, and certainly in many systems that require high-performance real-time control operations. In many cases, space and power constraints make feasible the implementation of such systems only when most of their logic is placed on a single silicon die (thus the name “system-on-chip”).

The objectives of this chapter are:

• To present an overview of the system-on-chip developed in this thesis as a general solution to the problem of high-performance real-time embedded control.

• To describe the system-on-chip solution implementation based on a dual processor architecture, which splits the control task into two parts: the high-performance control algorithm execution (rCSP) and the adaptation and interlocking process (embedded CPU).

• To describe the functionality and tailoring of the embedded CPU’s software
whose execution controls the overall system's behaviour.

- To present a Java-based tool designed for the on-line monitoring and commanding of the whole system-on-chip solution.

- Provide speed and complexity figures of the system-on-chip solutions' implementation.

### 6.2 Architecture Overview

System-on-chip designs have in general smaller size and lower power consumption than less integrated solutions [22], but lack a standardised design methodology. Moreover, the terminology "system-on-chip" only describes the fact that most of the functionality of the system, which in the past was probably scattered around a small Printed Circuit Board (PCB), is placed on a single chip. The gate densities achieved in current ASIC and FPGA devices provide designers with enough logic gates/elements to implement all the different functionalities on the same chip by mixing self-designed or customised modules with standard off-the-shelf ones. This possibility opens new horizons especially for embedded systems where space constraints are as important as performance, but it maybe also a source of non-reusable and complex designs that are difficult to debug/modify/upgrade. In order to avoid those problems, certain design decisions must be taken, bearing in mind that emerging industry standards will make maintenance and upgrading easier.

The architecture developed in this thesis (see Figure 6.1) is a single-chip multiprocessor embedded solution targeted for high-performance real-time control applications, especially those with extra requirements in terms of signal processing power, connectivity and reconfigurability.

Figure 6.1 presents the system-on-chip solution architecture, outlining the relation between the different elements of the system, namely the digital logic embedded in the system-on-chip, the analogue electronics required to drive the sensors' readings into the rCSP, and the actuators that transform the actions calculated by the rCSP into the proper actions to be performed upon the plant. Finally the plant itself is depicted,
Figure 6.1: Block diagram of a complete embedded control system outlining the developed system-on-chip architecture exemplified as a fly-by-wire airplane system (further discussed in Section 7.6).

The division of the tasks among the system modules is depicted in Figure 6.2 and summarised in Table 6.1. The architecture is divided into three sections, namely Control Kernel Section, Communication Section and General Section, although due to its scalability other IP cores can be added in the future if new requirements arise. The Control Kernel is in charge of executing and managing all control-related aspects. It comprises an embedded CPU and the rCSP processor interconnected by an AMBA compliant bus. Both processors run concurrently allowing the control process to be split into two: the control algorithm execution and the adaptation management. The former gathers all the required tasks for converting and sampling sensor inputs, executing the control law and sending the calculated action back to the system. This is efficiently performed by the rCSP. The adaptation management is run in the background by the embedded CPU in parallel with the control law execution. Periodically (at a significantly lower rate than the sampling one), it monitors the control state variables and/or the inputs/sensors information stored in the rCSP and, depending on their values and the application characteristics, it computes a new set of coefficients. The adaptation algorithm can take a variety of forms ranging from simple gain scheduling through to more advanced concepts such as self-tuning control. The impor-
tant point is that its functionality is separated from the high-speed execution of the main control law which uses the latest coefficient values provided by the adaptation.

![SoC functional diagram](image)

**Figure 6.2: SoC functional diagram**

The *Communication* section provides the system with the required support for the general data transfer with external devices common to most applications. The developed system-on-chip architecture includes a standard serial communication core (UART) for that purpose. Due to the nature of control applications the state variables (and other relevant information) are sent through the communication port to be monitored in real-time. The interlocking of the system is done by the CPU, which receives the external commands through the UART and directs the actions to be taken by the rest of the SoC components.

Analogue sensors and actuators as well as digital inputs and outputs are directly interfaced with the *rCSP* processor. There are many cases where the control system may need its inputs and outputs to be only digital (e.g. digital filtering, digital control stages, etc.). In that case the system complexity can be reduced significantly because no external analogue electronics (i.e. DAC and operational amplifiers) is needed and the *rCSP* can be configured without the SAC and PWM units. In other cases, the control system may require to interface only with analogue devices. Then, the pinning of the system will be highly simplified by configuring the *rCSP* with only PWM units.
for they are single bit modulated outputs.

The General section gathers the rest of the components (e.g. Timers and Memories) that are somehow common to most applications.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>rCSP Commanding</td>
<td>Embedded Microprocessor</td>
</tr>
<tr>
<td>rCSP Programming</td>
<td>Embedded Microprocessor</td>
</tr>
<tr>
<td>Coefficients Adaptation</td>
<td>Embedded Microprocessor</td>
</tr>
<tr>
<td>Algorithm Execution</td>
<td>rCSP</td>
</tr>
<tr>
<td>Signal Sampling</td>
<td>rCSP</td>
</tr>
<tr>
<td>Signal Modulation</td>
<td>rCSP</td>
</tr>
<tr>
<td>Internal Bus</td>
<td>AMBA (APB)</td>
</tr>
<tr>
<td>External Bus</td>
<td>AMBA (APB or AHB)</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Embedded Microprocessor</td>
</tr>
<tr>
<td>Serial Bus</td>
<td>UART IP Core</td>
</tr>
</tbody>
</table>

Table 6.1: Functionality division

The remaining of this section provides more details about the main architectural blocks of the system-on-chip solution.

6.2.1 Embedded CPU

The embedded processor commands the actions to be taken by the rCSP and performs certain tasks in parallel with the control law execution. Some of these tasks are the calculation of the appropriate set of coefficients, the monitoring and analysis of the system status and the management of external events. At booting time, the embedded CPU is in charge of configuring the rCSP, transferring to it the algorithm to be executed, its initial coefficients and setting up its register file. Since the computationally-intensive part of the control has been diverted to the rCSP and due to the numerical benefits of the controller formulation used by its software tools (see Chapter 7) a small embedded processor or micro-controller core would be adequate to perform its time-constrained tasks with no delay. Fixed-point precision should be
sufficient in most cases, which further decreases the computational requirements of the embedded CPU. The adaptation of the coefficients (the CPU's most demanding task) is done on a typical real-time application at a relatively low frequency, typically in the order of tens of Hz, which doesn't present any difficulty even for the simplest CPU architecture. Some examples of re-configurable CPUs easily synthesisable on an FPGA technology are: (i) LEON [13], an open source RISC microprocessor SPARC v8 compatible, supported (amongst others) by the European Space Agency (ESA); (ii) Altera's Nios [3] (for Altera devices); (iii) Xilinx's MicroBlaze [50] (for Xilinx devices); (iv) Zilog's Z80 soft IP core, a mature processor that is still one of the leaders in the 8-bit market.

6.2.2 rCSP processor

The rCSP processor is in charge of performing the real-time tasks inherent to the control algorithm such as sampling, control law execution, actions modulation and data outputting. Its architecture and implementation were described in Chapter 5. Its standard configuration is designed to suit most of the general real-time control cases. However, its main parameters (e.g. storage capacity of every module, number of SAC or PWM units required, buses, etc.) can be easily tailored to fit the requirements of a particular application, hence avoiding a waste of resources and increasing performance.

6.2.3 Memory System

As shown in Figure 6.1, the proposed system-on-chip solution requires a certain amount of memory storage. Typically, FPGAs offer a certain number of SRAM memory blocks distributed among their logic cells. This memory is used for implementing the rCSP's memory units as well as a small ROM to boot the embedded CPU. The embedded software has to be initially stored in a non-volatile memory but, although some FPGA technologies offer a limited amount of such memory, in general a non-volatile external memory (usually FLASH memory) containing the embedded software is attached to the system and interfaced with the embedded CPU. However, this memory technology is relatively slow and would reduce the overall system performance if it were used as
main memory. Therefore a third memory technology is included (usually SDRAM or DDRAM) to be operated as the embedded CPU's main memory.

6.2.4 External Communication

A *Universal Asynchronous Receiver Transmitter* (UART) soft core is included to provide serial communication capabilities between the system-on-chip solution and external devices, because it requires less complexity in terms of hardware and software than for example an *Ethernet* or a *Universal Serial Bus* (USB) interface, yet offers enough performance. Another advantage is the fact that every computer (and many other devices) contains this kind of serial communications capability, assuring the connectivity of the proposed system-on-chip solution.

6.3 LEON-Based System-on-Chip Implementation

The architecture of the system-on-chip solution presented in this thesis, although technology-independent, has to be eventually implemented using a particular fabric technology and embedded CPU model. As proof of concept and due to the resources available, the first implementation was based on an Altera Excalibur FPGA [2], which includes a hardwired ARM922T microprocessor [1]. However, the poor results obtained in terms of performance and Excalibur Board related problems favoured the use of other FPGA technologies and CPU architectures (see Table 6.2).

<table>
<thead>
<tr>
<th>Field</th>
<th>Vendor</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Technology</td>
<td>Altera</td>
<td>Excalibur EPX10</td>
</tr>
<tr>
<td></td>
<td>Xilinx</td>
<td>Spartan2E XC2S600E, Virtex2 XC2V1000</td>
</tr>
<tr>
<td>Embedded CPU</td>
<td>ARM</td>
<td>ARM922T (hardwired in the Excalibur FPGA)</td>
</tr>
<tr>
<td></td>
<td>Gaisler Research</td>
<td>LEON 2 (open source version)</td>
</tr>
</tbody>
</table>

Table 6.2: Different implementation architectures used during the development and implementation of the system-on-chip architecture presented in this chapter

In particular, the embedded CPU of choice was the open-source version of the LEON processor [13], and therefore the implementation described in this chapter will
be based on this CPU architecture. In this thesis, the \textit{LEON2 XST} (open-source) processor's version is used. Throughout the rest of the thesis, the version code will be omitted and \textit{LEON2 XST} will be therefore referred as \textit{LEON}. The general description presented in the following lines regarding the \textit{LEON} processor, its licensing and performance is based on the \textit{Leon2 Processor User's Manual. XST Edition} [13].

\section*{6.3.1 \textit{LEON} Overview}

The \textit{LEON} VHDL model implements a 32-bit processor conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications with the following features on-chip (see Figure 6.3): separate instruction and data caches, hardware multiplier and divider, interrupt controller, debug support unit with trace buffer, two 24-bit timers, two UARTs, power-down function, watchdog, 16-bit I/O port, flexible memory controller, ethernet MAC and PCI interface. New modules can easily be added using the on-chip AMBA AHB/APB buses. The VHDL model is fully synthesizable with most synthesis tools and can be implemented on both FPGAs and ASICs. The \textit{LEON} processor is developed by Gaisler Research and it is currently supported (amongst others) by the European Space Agency (ESA). Due to its open source licensing, \textit{LEON} benefits from the testing, feedback and cooperation of a wide community of users. Many companies and universities are currently developing different \textit{LEON}-based hardware and software projects (see Gaisler Research web page for an updated list of projects and partners). Some of these projects include the porting of Linux and Java for this architecture, the development/improvement of different modules such as floating-point units (FPU), cache units, etc.

\textbf{\textit{LEON} License}

The original fault-tolerant (FT) \textit{LEON} design (\textit{LEON-FT}) is commercially available from Gaisler Research. It includes advanced fault-tolerance features to withstand arbitrary single-event upset (SEU) errors without loss of data. The fault-tolerance is provided at design (VHDL) level, and does not require an SEU-hard semiconductor process, nor a custom cell library or special back-end tools.
The LEON VHDL model is an open source release of LEON-FT with all the FT logic removed. The LEON VHDL model is provided under two licenses: the GNU Public License (GPL) and the Lesser GNU Public License (LGPL). The LGPL applies to the LEON model itself while the remaining support files and test benches are provided under GPL. This means that you can use LEON as a core in a system-on-chip design without having to publish the source code of any additional IP-cores you might use. You must however publish any modifications you have made to the LEON core itself, as described in LGPL.

Performance

Using the Dhrystone 2.1 benchmark, LEON executes around 1,500 dhrystones/s/MHz. This translates to roughly 0.85 dhrystone MIPS/MHz. Configurations with hardware mul/div and larger caches usually perform somewhat better. The dhrystone benchmark was compiled with LECCS-1.1.5 (gcc-2.95.3) using -O2 optimisation. Dhrystone is a benchmark program that tests a system's integer performance. The program is CPU bound, performing no I/O functions or operating system calls.
LEON Configuration

The LEON model is highly configurable, allowing the model to be customised for a certain application or target technology. LEON's functionality can be easily extended by the integration of new modules through its on-chip AHB/APB Bridge (see Figure 6.3). The LEON processor model provides an interface to the high-performance GRFPU (available from Gaisler Research), the Meiko FPU core (available from Sun Microsystems) and the incomplete LTH FPU. A generic co-processor interface is provided to allow interfacing of custom co-processors (CP). In that way for example, a custom vector unit could be interfaced as a co-processor providing SIMD processing capabilities to the LEON model.

6.3.2 Tailoring LEON for the system-on-chip control solution

LEON includes some particularly attractive features relevant to the purpose of this thesis: its model is vendor-independent (i.e. it synthesises in different FPGA and ASIC models), its design is open source and freely customisable, and there is a large community working with it, sharing knowledge, providing support and expanding its functionalities.

In order to use LEON as the embedded CPU for the SoC solution proposed in this thesis, first its architecture must be tailored in order to reduce its size, excluding all the modules that are not strictly necessary to perform the tasks presented in Table 6.1. Such modules are:

- **FPU** (floating-point unit). The only numerically demanding task associated to the embedded processor is the adaptation of the coefficients. This can be easily performed using fixed-point arithmetic and therefore no floating-point unit is needed. Note that an FPU requires about the same silicon space as the whole Integer Unit of the processor.

- **CP** (co-processor) interface. The SoC solution doesn’t require a co-processor interface because the rCSP is not a co-processor but an independent processing unit periodically supervised by LEON.
• **PCI, Ethernet.** The SoC solution interfaces with external devices by means of Serial Communication via LEON’s UART.

• **Debug modules.** These modules are not required in the final version of the SoC solution. The embedded processor tasks are neither complex nor computationally demanding, and there is no operative system running on it. This scheme provides a higher level of predictability in the processor’s execution while reducing the risk of errors or bugs prone to large scale software systems.

**Hardware Interface between LEON and rCSP**

After producing a minimal version of the LEON processor, the second stage is to interface it with the rCSP. This is seamlessly achieved by means of the rCSP’s built-in APB slave interface (see Figure 6.4). In order to make LEON aware of the presence of a new module connected as an APB slave, this module should be given a new address and listed in LEON’s APB mapping table.

The LEON VHDL model provides a number of system support functions directly on-chip (e.g. cache controller, UART controller, timers controller, Interrupt controller, etc.), which are register mapped and can be accessed through the APB bus according to a mapping table managed by the LEON’s module apbext.vhd. LEON’s model defines the addresses range from Ox80000000 to Ox8FFFFFFF to be used for APB mapped modules. The APB accessible system support functions provided in the LEON model are mapped between the addresses Ox80000000 and Ox800000CC and therefore a higher address should be used to reference the rCSP. As seen in Section 5.5, the base address of the rCSP should be a multiple of Ox4000. In this case the first available address that fulfils all the requirements is Ox80004000. The following lines outline LEON’s APB mapping table once configured.

- Ox80000000 - (LEON) Memory configuration register1
- Ox80000004 - (LEON) Memory configuration register2
- ...
- Ox800000CC - (LEON) DSV UART scalar register
- Ox80004000 - rCSP v3.0 Base Address

In order to access a particular rCSP module LEON will generate the proper ad-
address as \textsf{rCSP Base Address + Module's Relative Address}. The addresses of the different \textsf{rCSP} modules were described in Section 5.5.

![Diagram of System-on-Chip implementation using LEON2 processor as embedded CPU]

\textbf{Figure 6.4: SoC implementation using LEON2 processor as embedded CPU}

\section*{6.3.3 Final Remark}

Since the performance and functionality offered by the LEON processor as the SoC's embedded CPU are significantly higher than the actual requirements found in common control systems and in some special cases (e.g. particularly power aware systems), its size or consumption may not be appropriate. In this research the LEON processor has been chosen as proof of concept and to demonstrate the actual integration between an embedded CPU and the \textsf{rCSP} although the system-on-chip architecture here presented is open to any other embedded CPU technology. However, a LEON-based architecture provides a scalable solution capable of fulfilling the ever-increasing functionalities and performance required by current and future control projects, where for example a number of \textsf{rCSP} units could run in parallel, all of them being commanded, monitored and adapted by LEON. Furthermore, the open-source nature of LEON as well as the support offered by Gaisler Research and the world-wide academic commu-
nity make it the embedded processor of choice in numerous research and commercial projects.

6.4 Embedded Software

Embedded systems are often implemented using self-sufficient code (i.e. it doesn’t require an operating system), taking control of the embedded computer from power-up ([38],[8]). Although there is a wide variety of real-time operating systems (OS) targeted for embedded applications (e.g. pSOS and VX Works from Wind River, Windows CE from Microsoft, Palm, embedded Linux), the developed software architecture is implemented as a stand alone (self-sufficient) application. Operating systems provide the programmer with different services and higher-level functions (e.g. multi-task management, device interfacing, etc.) that facilitate the creation of complex software systems for a particular CPU architecture. However, the system-on-chip solution developed in this thesis is targeted for high-performance real-time embedded control applications and therefore it has to balance certain aspects such as reliability, predictability, resources and performance. Since the hard real-time processing is diverted to the rCSP, the embedded CPU’s software plays mainly a managerial role and therefore most of the functionalities provided by an OS are superfluous and its overhead in terms of memory footprint inappropriate. Furthermore, part of the objectives of this research was to develop a low-cost and vendor-independent solution in terms of implementation technology (i.e. FPGA or ASIC) and architectural components (i.e. CPU model and its related software), that highly differ from the costs and the architecture dependency that an operating system imposes.

There are two different types of software running on the developed system-on-chip solution. The first one contains the control algorithm and is executed on the rCSP. This software is generated automatically by the software tool that will be presented in Chapter 7, directly from the block diagram descriptions of the control algorithm. The second type of software runs on the selected embedded CPU and it plays a more managerial role being in charge of booting and configuring the CPU, booting and commanding the rCSP, monitoring the system, adapting the control
coefficients, and communicating the system through the serial port.

This section describes the software executed on the embedded CPU, its modules and functionalities, discussing also its portability to other platforms. First an overview of the whole software system is provided, analysing which modules are application-dependent, which modules are CPU dependent and which ones should not be modified. Afterwards, a detailed functional description of the software system is presented. A number of libraries developed for simplifying the system-on-chip solution programming are introduced in this section. Appendix C provides a detailed description of all them.

6.4.1 Architecture Overview

The list of software modules that are executed on the embedded CPU is shown in Table 6.3. The number of stars (*) on the left hand side of a module's name represents the degree of dependency (as opposed to portability) that this particular module has. A modules whose name starts with "cpu_" is, at some level, CPU-dependent. The assembler file cpu_boot.s (★★★) is absolutely dependent on the selected CPU architecture and therefore it is not portable at all, although as explained before it is usually provided by the CPU vendor/manufacturer. The cpu_irq.c (★★) module sets up and configures the Timer's and UART's interrupt services. This module is CPU dependent and peripheral (Timer,UART) dependent, although its porting is relatively simple because it only involves some basic register configuration. The header cpuamba_map.h (*) defines the memory-mapped peripherals addresses and therefore depends on the CPU and the peripherals supported by it. However, its porting is usually straightforward since most architectures support, in a similar way, the basic peripherals required in the proposed SoC solutions and its porting would only require the addition or modification of the peripherals' mapped addresses and related bit masks. The rest of the modules are CPU independent and therefore can be ported from one platform to another without any modification.

Only the modules marked with the symbol (☆) in Table 6.3 are usually affected when the system-on-chip solution is reused in different control projects. In this case, the designer should only modify some sections of these modules in order to tailor the oth-
Chapter 6: System-on-Chip Solution for Embedded Real-Time Control

<table>
<thead>
<tr>
<th>File</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcspv3.h</td>
<td>rCSP hardware constants and its memory-mapped modules (addresses, masks, etc.)</td>
</tr>
<tr>
<td>app_name.h</td>
<td>Application specific parameters</td>
</tr>
<tr>
<td>cpu_mem.h</td>
<td>Peripherals' mapping and CPU related constants (including IRQ numbers, register bit masks, etc.)</td>
</tr>
<tr>
<td>cpu_boot.s</td>
<td>CPU boot routine</td>
</tr>
<tr>
<td>cpu_irq.c</td>
<td>Initialisation and management of Timer1 and UART IRQ services</td>
</tr>
<tr>
<td>main.c</td>
<td>Entry point to the C executable</td>
</tr>
<tr>
<td>cnt1_data_filters.c</td>
<td>Different filter routines</td>
</tr>
<tr>
<td>cnt1_adaptation.c</td>
<td>Different adaptation routines</td>
</tr>
<tr>
<td>rcspv3_boot.c</td>
<td>Routines for booting the rCSP</td>
</tr>
<tr>
<td>rcspv3_mem.c</td>
<td>Routines for accessing the different rCSP's memory modules</td>
</tr>
<tr>
<td>console.c</td>
<td>Routines for sending data to the console and Java monitoring tool</td>
</tr>
</tbody>
</table>

Table 6.3: Embedded software modules

otherwise general solution for a particular project. For instance, the header app_name.h (\(\ast\)) defines some parameters specific to a particular application, such as the coding of the different commands transmitted through the UART to the embedded CPU. This module has no stars because it depends only on the application regardless of the CPU or address-mapping, hence it is fully portable. An example of the contents of this module for a MAGLEV vehicle suspension control application is presented below:

/* UART Commands */

#define APP_UART_CMD_SWITCH_ON 's'
#define APP_UART_CMD_SWITCH_OFF 'o'
#define APP_UART_CMD_LEVITATE 'l'
#define APP_UART_CMD_DELEVITATE 'd'

Files cnt1_adaptation.c (\(\ast\)) and cnt1_data_filters.c (\(\ast\)) can be used as templates to build a particular adaptation scheme or data filtering routine. They include, respectively, a scheduling algorithm for adapting the rCSP coefficients and a window average data filter routine used during the adaptation to filter the sensors' readings. The file main.c (\(\ast\)) contains the command loop, which is in charge of interpreting and executing the commands received through the serial port, and it will be described later in more detail. The interpretation of the commands is done by means of the constants defined in the header app_name.h (\(\ast\)).
Chapter 6: System-on-Chip Solution for Embedded Real-Time Control

The unmarked files listed in Table 6.3 (with the exception of console.c) are generated and modified by the software tool that will be presented in the next chapter, and should not be modified by the control designer. The header rcspv3.h contains rCSP hardware-related constants used by different routines that access rCSP's register-mapped components. The files rcspv3_boot.c and rcspv3_mem.c contain the drivers that interface the rCSP processor with the selected CPU. They include routines for transferring the control algorithm and coefficients into the rCSP, as well for reading the state variables and sensors' values from it. These modules are therefore dependent on the rCSP architecture and on the control algorithm.

This software architecture and its code generation scheme allows control designers to focus on the particularities of each project (e.g. system commands, adaptation techniques, etc.), while the typical time-consuming and error-prone low-level interfacing and optimised control law generation are automated (see Chapter 7).

6.4.2 Functional Description

The software running on the embedded CPU can be divided into three functional parts: initialisation, application-dependent configuration and algorithm execution. The functional diagram of the embedded software developed for the SoC solution proposed in this thesis is depicted in Figure 6.5. The first part is a small assembly routine (cpu_boot.s), in charge of booting the CPU by configuring some basic systems such as memory table and cache, and transferring the rest of the software to the main (faster) memory. After the configuration process is finished, this routine jumps to the first instruction of the main module (main.c), which defines the entry point for the C code section. The application-dependent configuration is written in ANSI-C and its main duties involve the interrupt service routines (ISR) implementation and peripherals' configuration for a particular application. The algorithm execution refers to the piece of software that is in charge of managing the control activities by commanding the rCSP, adapting periodically the control coefficients and communicating the system through the serial port.

This software architecture provides control designers with:
Chapter 6: System-on-Chip Solution for Embedded Real-Time Control

1. A general structure valid for a wide variety of high-performance real-time embedded control applications, supported by a number of templates and a software generation tool (see Chapter 7).

2. An easy to understand, tailor and upgrade embedded software architecture.

The basic structure of the main C function, which is the software’s entry point after booting the CPU, is outlined below.

```c
int main(void) {
    char ch;
    IL * (volatile unsigned int) SOC_RCSFV3_TIMER_BASE;
    // UART Set Up
    irq_uart_init();
    console(INFO, "----------- Booting Sequence -------------\r\n");
    rCSFv3_BootUp();
    // Adaptation Timer Set Up
    irq_timer1_init(); // Timer1 IRQ Setup
    catch_interrupt(timer1_irq_handler, TIMER0_IRQ); // Handler for Timer1 IRQs
    setup_adaptation_time(my_time.ns);
    console(INFO, "----------- System Ready for Control --------------\r\n");
    while (true) {
        ch=getchar();
        // Commanding Loop Here...
    }
}
```

A detailed functional description of the software execution flow is provided below.

![Figure 6.5: Embedded software boot process flowchart](image)
Booting and configuring the embedded CPU

There are different approaches to boot an embedded system. A common option is to place all the embedded software (with the boot code at the beginning) in a non-volatile memory (typically FLASH memory) mapped at address 0x0. Due to the slow read (and especially write) speeds common to all FLASH memories, the boot code usually includes a small section in charge of copying the software placed on the flash memory (with exception of the boot code) into a faster memory technology (usually SDRAM or DDRAM). Once this process is finished the boot code allows the processor to branch to the entry point of the newly allocated code.

In the particular case of the proposed system-on-chip architecture, the boot code is hardwired and implemented with distributed logic cells as a ROM together with the rest of the VHDL modules. The reason for doing that is fourfold. First, it increases the booting speed since the code is directly implemented using FPGA's logic cells and SRAM memory elements. Second, the boot code is not intended to be modified unless the design is affected by important architectural changes, and therefore there is no need for it to be placed together with the rest of the software. Third, in case such architectural changes happen, it is likely that some parts of the design will be reconfigured and therefore a new bitstream will be downloaded into the FPGA, containing also a modified version of the boot ROM. Fourth, the boot code footprint is less than 1 kilobyte, which doesn't compromise the resources of even the most modest modern FPGA devices.

After the embedded CPU is initialised, and the embedded software is placed in a fast memory technology, the boot code branches the embedded CPU's programme counter (PC) to the entry point (main.c()) of the software's application dependent configuration section (see Figure 6.5), where the peripherals (including the rCSP) and the ISRs are set up.

Booting and Testing the rCSP

During the application dependent configuration sequence of the embedded CPU, this processor transfers the real-time algorithm and the initial set of coefficients to the
rCSP, configuring also its internal registers. This sequence is depicted in Figure 6.6. In the optional test sequence (see Figure 6.7), the embedded CPU boots the rCSP with a test algorithm and coefficient set, lets it run for a number of predefined cycles and then analyses the algorithm internal variables, comparing them with the expected values. If the test is not successful then an error message is outputted through the UART and the system execution is cancelled.

![Figure 6.6: rCSP's boot sequence performed by the embedded CPU](image)

![Figure 6.7: rCSP's test sequence performed by the embedded CPU](image)

**Commanding loop**

Once the processors and peripherals have been successfully booted and configured, the software enters into the *commanding loop* (see Figure 6.8). The commands are usually introduced through a console device (e.g. a keyboard) and sent to the embedded CPU via UART. Then, depending on these inputs, a particular set of commands are sent to the rCSP such as RUN, STOP, RESET, BOOT, READ_SENSORS, etc. These standard commands are already implemented but the control designer can implement new commands for a particular application. The implementation of a new command is done by introducing a new *case* statement in the commanding loop.
The following code sketches a section of the actual commanding loop located in the main.c file:

```c
while (true) {
    ch=getchar();
    switch (ch) {
    case c_IL_CMD_START:
        \ Commands the rCSP v3.0 to start the execution of
        \ the control algorithm and starts the adaptation Timer
        \ *IL = CMD_RUN; *IL = CMD_NOP;
        Timer1_Start(); break;
    case c_IL_CMD_STOP:
        \ Commands the rCSP v3.0 to stop the execution of
        \ the control algorithm and stops the adaptation Timer
        \ *IL = CMD_STOP; *IL = CMD_NOP;
        Timer1_Stop(); break;
    case c_IL_CMD_READ_SENSORS:
        \ Commands the rCSP v3.0 to retrieve the current reading
        \ of a particular sensor
        Console_Data(Read_Sensor(..)); break;
    case : ...
    otherwise : ...
    }
}
```

As can be seen, the structure of the commanding loop is very clear and the standard functionalities are already included. The constants associated with each command (e.g. c_IL_CMD_STOP) are defined in the header file app_name.h.

Without losing generality the commanding loop has been implemented by polling the UART. This simplifies the software structure and serves as proof of concept. Polling can be implemented in different ways but perhaps the easiest one is to use the C function get_char(), which in embedded CPU technologies is usually inter-
faced with the UART. Otherwise, it is simple to read directly the UART registers. Appendix xxx provides a basic introduction to the programming techniques required to access register-mapped peripherals in C.

Reading Sensor and State Variables

In order to access the sensors (or digital inputs) values and state variables stored in the rCSP the following functions are provided:

```c
struct IRAM_Reading Read_Dual_Sensors(struct IRAM_Request iram_req)
unsigned int Read_Sensor(unsigned int sensor_nr)
unsigned int Read_Single_STV(unsigned int addr)
```

As the names suggest, the first one retrieves two different sensors’ (or inputs’) readings together, the next one returns only one sensor reading, while the last one retrieves the value of a particular state variable. The reason to retrieve two sensors’ readings simultaneously is that often the decision that triggers the adaptation process depends on the ratio of two different sensors’ values. Retrieving two values at the same time is possible since the input/output data word length (usually 8, 10, or 12-bit) is much smaller than half the size of the AMBA data bus. A more detailed description of these functions together with their related structures can be found in Appendix C.

Adaptation

The adaptation process is executed at regular intervals triggered by a Timer interrupt. When such interrupt arrives, the processor starts the execution of the adaptation routine depicted in Figure 6.9. In order to define how often the adaptation process should be executed, the following function (defined in the module cpu_irq.c) is used to set up the adaptation frequency:

```c
void setup_adaptation_time(int time_ns)
```

As mentioned before, a gain scheduling scheme is implemented in the module cntl_adaptation.c. Different adaptation schemes can be added into this module and
the selected one(s) should be called within the Timer1's interrupt handler (located in the cpu_irq.c module). In its simplest form this handler has the following code:

```c
void timer0_irq_handler(void) {
    My_Adaptation_Method_Routine(); \ defined in cntl_adaptation.c
}
```

Often, the sensors' readings are filtered (e.g. averaged over a particular period of time) before being used. Two window average filtering routines are available in the module cntl_data_filters.c and can be used as a templates to implement new ones. The structure of a generic adaptation routine is presented in Figure 6.10 (scales are not real). The adaptation routine is periodically called every $T_a$ seconds. Assuming, for instance, an adaptation frequency of 10 Hz, this diagram depicts the different actions taken during the Timer interrupt execution, which should last less than $T_a = 0.1$ seconds ($T_a[k] = k \cdot T_a$). In general, when the Timer's interrupt arrives the CPU prepares to jump to the interrupt handler code. Once the execution of the adaptation routine starts it reads some sensors' values and/or state variables stored on the rCSP and sends them to the UART for monitoring purposes. Then it processes them (e.g. applying some kind of filtering) and upon the results obtained it decides whether the operating point has changed or not. If the operating point remains the same, the adaptation routine finishes. Otherwise, a new set of coefficients is either selected (gain scheduling) or calculated in real-time and as soon as they are ready they are transferred to the rCSP, using the commands described in Chapter 5.

![Figure 6.9: Generic adaptation algorithm called periodically by the Timer's IRQ](image-url)
6.4.3 LEON-specific software issues

The following lines briefly summarise the main software issues particular to the LEON architecture and cross-compiler system.

- **Cross-compiler system:**
  LECCS is the cross-compiler system provided with the LEON processor. Detailed information about it can be found in [44]. LECCS extends the gcc compiler to support the LEON and ERC32 processors.
  Syntax example:
  
  ```
  sparc-rtems-gcc -O2 -mv8 cntl_data_filter.c
  ```

- **Interrupt installation:**
  In LECCS, the standard C-library has been extended to include interrupt support. The function to be used as interrupt handler is assigned through the function call:
  
  ```
  extern void *catch_interrupt(void func(int irq), int irq);
  ```
  An example:
  ```
  catch_interrupt(timer1_irq_handler, TIMER1_IRQ);
  ```

- **Accessing the UART.**
  The system’s stdin/stdout (standard input/output) are mapped into the UART A. Therefore this UART can be accessible via the usual C stdio functions (e.g. `getchar()`, `printf()`, etc.).
6.5 Monitoring and Commanding the System-on-Chip solution

In order to provide a graphical interface for commanding and monitoring the SoC solution, a Java-based on-line tool has been developed. This tool:

- Has been developed with 100% Sun's Java compatible code, allowing its portability to any computer system that supports Java2 and serial communication.
- Allows the graphical monitoring of the selected state variables' and sensors' evolution.
- Allows the commanding of the system-on-chip solution.
- Makes it possible to parameterise different aspects of the communication (ports, speeds, etc.) and the graphical representation.
- Includes a console where the text messages sent by the embedded software are displayed. This messages can be used, for instance, to record the system's activity or to communicate with a human operator.

The tool differentiates between a console message and a value to be graphically displayed by the structure of the information sent through the UART. The latter has the following structure (each element in brackets has 1 byte length):

(Identifier)(Variable Number)(High-bit Value)(Low-bit Value)

The brackets used above are introduced for clarity and they are not actually transmitted. The default Identifier is the @ symbol. The Variable Number is an index that links a particular display window with a particular state variable. Since UART based serial communication is byte-based, and the values sent by the CPU are 16 bits wide, they are first split into two bytes and the transferred in the order previously shown. By inspecting the module console.c the actual details of such a process can be found.
6.6 System-on-chip implementation figures

The implementation of the system-on-chip solution has been performed with the rCSP's standard configuration, using the FPGA technologies listed in Table 6.4. Appendix A presents an overview of the main architectural parameters of these devices. Leon has been implemented with 4 kbytes direct-mapped instruction and data caches with line size 8 and random replacement policy. The synthesis tool used was Synplify Pro v.7.3.1 (Built: Jun 2 2003) from Synplicity, and the place & route tool used was Xilinx's ISE (Integrated Software Environment) v.6.1.

Any data sequence sent through the UART that does not start with the @ identifier, is understood as a console message. For example, if the data sequence

@ 0x01 0x00 0xFF

is received through the UART, the tool interprets it as a value (0x00FF=255) to be graphically displayed on the window number 1. On the other hand, a data sequence such as ‘‘This is a console message’’ will be just printed out on the console window.
Table 6.4: Targeted FPGAs

Speed figures

The post place & route clock frequencies obtained for the targeted FPGA technologies are summarised in Table 6.5. Note that LEON and rCSP’s core are clocked at the same frequency.

Table 6.5: rCSP’s clock frequencies

Table 6.6 quantifies the sampling frequencies achievable by the system-on-chip solution for a particular FPGA technology. The control algorithms implemented are the same ones used in Section 5.8.

Table 6.6: Maximum sampling frequency figures
Complexity figures

The complexity figures of the system-on-chip solution are summarised in Table 6.7. As can be seen, the whole system-on-chip solution can be successfully implemented in low-cost FPGA technologies such as Spartan-IIIE.

An RTL view of the system-on-chip solution generated by the synthesis tool is presented in Figure 6.12.

<table>
<thead>
<tr>
<th>Logic Utilisation</th>
<th>Spartan-IIIE</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Flip Flops</td>
<td>2,568 (18%)</td>
<td>2,561 (25%)</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>7,622 (55%)</td>
<td>7,116 (69%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic Distribution</th>
<th>Spartan-IIIE</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of occupied Slices</td>
<td>5,556 (80%)</td>
<td>5,118 (99%)</td>
</tr>
<tr>
<td>Total 4 input LUTs</td>
<td>9,754 (70%)</td>
<td>9,205 (89%)</td>
</tr>
<tr>
<td>Used as logic</td>
<td>7,622</td>
<td>7,116</td>
</tr>
<tr>
<td>Used as a route-thr.</td>
<td>196</td>
<td>153</td>
</tr>
<tr>
<td>Used for Dual Port RAMs</td>
<td>1,936</td>
<td>1,936</td>
</tr>
</tbody>
</table>

| Number of bonded IOBs     | 224 (68%)          | 226 (68%)          |
| Number of Block RAMs      | 30 (41%)           | 14 (35%)           |
| Number of MULT18X18s      | N/A                | 2 (5%)             |
| Number of GCLKs           | 2 (50%)            | 2 (12%)            |

Table 6.7: System-on-chip solution's complexity figures for the selected Xilinx devices

LEON complexity and speed figures

In order to give an idea of the resources used by the LEON processor (as configured in Figure 6.4) and the performance that it can achieve on a particular FPGA technology, the following post place & route figures are provided. They correspond to the LEON processor implemented alone on the selected FPGA device. The clock frequency figures are presented in Table 6.8, while the complexity figures are provided in Table 6.9.
Chapter 6: System-on-Chip Solution for Embedded Real-Time Control

<table>
<thead>
<tr>
<th>Technology Part</th>
<th>Clock (MHz)</th>
<th>Clock (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan IIE XC2S600E</td>
<td>36.3</td>
<td>27.513</td>
</tr>
<tr>
<td>Virtex2 XCV1000</td>
<td>80</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Table 6.8: Clock frequencies for LEON

<table>
<thead>
<tr>
<th>Spartan-IIE XC2S600E</th>
<th>Virtex-II XCV1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilisation</td>
<td></td>
</tr>
<tr>
<td>□ Number of Slice Flip Flops</td>
<td>1,546 (11%)</td>
</tr>
<tr>
<td>□ Number of 4 input LUTs</td>
<td>5,185 (37%)</td>
</tr>
<tr>
<td>Logic Distribution</td>
<td></td>
</tr>
<tr>
<td>□ Number of occupied Slices</td>
<td>3,653 (52%)</td>
</tr>
<tr>
<td>Total 4 input LUTs</td>
<td>6,435 (46%)</td>
</tr>
<tr>
<td>... used as logic</td>
<td>5,185</td>
</tr>
<tr>
<td>... used as a route-thr.</td>
<td>98</td>
</tr>
<tr>
<td>... used for Dual Port RAMs</td>
<td>1,152</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>215 (66%)</td>
</tr>
<tr>
<td>Number of Block RAMs</td>
<td>18 (25%)</td>
</tr>
<tr>
<td>Number of MULT18X18s</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of GCLKs</td>
<td>1 (25%)</td>
</tr>
</tbody>
</table>

Table 6.9: LEON's complexity figures for the selected Xilinx devices

6.7 Summary

This chapter presented the system-on-chip control solution developed in this thesis. Due to the device-independent nature of the architecture, its implementation was described assuming the LEON processor as embedded CPU. The particularities of the embedded software architecture and the system-on-chip interfacing were also addressed. The chapter concluded with the system-on-chip implementation figures in terms of speed and complexity.
Figure 6.12: Annotated RTL view of the system-on-chip solutions
Chapter 7

rCSP-based Design Methodology and rCSP Development Suite

7.1 Introduction and Objectives of the Chapter

It is been said that research in any area is 1% inspiration and 99% perspiration. A similar remark applies to digital control. In particular, a real control problem usually involves 49% understanding the process and objectives, 49% creating a suitable software environment for control, and only 2% calculating the control law [39]. This traditional working method could be improved if the 49% of time expended in the hardware/software co-design process could be automated.

This chapter presents the methodology and design flow developed in this thesis for designing and implementing rCSP-based embedded real-time digital control systems. Such a design flow automates most of the intermediate steps required to transform a system description into a fully functional hardware-software implementation. This will be illustrated by two implementation examples, namely a digital flight controller and a hard disk drive (HDD) controller that includes Kalman filtering to provide an estimation of the HDD's head position in presence of measurement noise.

The objectives of this chapter are to:

- Introduce the methodology and design flow developed in this thesis.
• Clarify this methodology and demonstrate its efficiency, by presenting two detailed design examples.

• Present the rCSP Development Suite which is designed to support and automate the different design-to-implementation stages of a rCSP-based control system.

7.2 Traditional Approach

Five main steps can be identified in the process of designing and implementing an embedded digital control system.

1. The system is modelled and an appropriate control law is obtained by means of suitable control design techniques.

2. The controller is discretised (if designed in the s-plane) and its real-time equations extracted.

3. These equations and other software routines (such as communication between modules, output modulation, register access, etc.) are encoded (usually in C and assembly language) for a particular processing technology such as DSP, MCU or microprocessor. Sometimes, especially in high-performing real-time applications, the computational load requires the use of some extra hardware units (such as vector units, coprocessors or parallel architectures) in order to achieve the required sample rates.

4. The hardware-software platform is simulated and tuned-up until the required performance is achieved.

5. The control system is physically implemented on a Printed Circuit Board (PCB) where the different elements are interconnected. The PCB is tested for physical and lay-out problems and after the required reworking is embedded into the system to be controlled.

These steps involve a highly iterative process, where many different factors are at play. Furthermore, the programming of the system and its tuning-up requires...
the design of hand-crafted application- and architecture-dependent software modules, usually in a mixture of C and assembler. As can be seen, this approach requires highly interdisciplinary skills in terms of hardware, software and real-time control co-design. Furthermore, the amount and complexity of the software that needs to be written (often architecture dependent) is usually an error-prone task that will lead to a later debugging process. Different control solutions require different levels of performance which often imply the use of different processing devices (depending on the application and budget) from different vendors and families. This translates into different PCB layouts, architectures with different pinning, different voltages, and different low-level programming issues among others.

7.3 Design-to-Implementation Methodology

Chapter 5 presented the rCSP, an application specific processor targeted for high-performance embedded real-time control. However, the processor alone does not relieve control designers from the above-described tedious and error-prone system programming and tuning tasks. Therefore, one of the main goals of this research is to analyse the feasibility and role of automated (thus error-free and highly optimised) software and hardware generation, in other words, to study how much of the design-to-implementation load can be taken away from the control designer by a suitable software tool made to support the development of rCSP-based control systems. As a result, this chapter presents a design-to-implementation methodology based on the rCSP Development Suite, a software tool developed in this thesis that allows the programming of the system directly from a controller block diagram description, thus automating steps 2,3 and 4 presented in previous section. Its interlinked iterative design process allows any changes in the hardware, software or control parameters to automatically adjust the rest of the system.

The steps required in the design-to-implementation methodology proposed in this thesis are outlined below and summarised in Figure 7.1:

1. System Description. In many cases, the design of a controller is not a time
Chapter 7: Design Methodology

demanding task. In some cases, this controller already exists (usually in analogue form) and it must be converted into a digital format. If high sample rates are required, a delta operator based controller description would often provide a more numerically robust solution, although in many cases the control designer may only be proficient in using the shift-operator based transform. Whatever the case is and the controller internal description, the rCSP Development Suite takes this description and automatically generates the real-time control equations (in delta form), calculates the coefficients and cross-compiles such equations into rCSP's assembly and machine code.

2. Control law validation. Once the control law is generated, the control system is simulated (for example in the s-plane) in Simulink, using traditional Simulink libraries.

3. Real-time equations validation. Then, the controller is replaced by the soft version of the rCSP model which uses MATLAB's floating-point arithmetic. The system is simulated again and its response is compared with the original one. Mismatches between the original system's outputs and the real-time ones are most likely due to some mistake in the parameters given to the software tools such as desired sampling frequency or number of control inputs. However, numerically ill-conditioned controllers would also produce a mismatch.

4. Coefficients' quantisation validation. Once the real-time equations have been validated, the next step is to validate the coefficient quantisation by selecting them as active coefficients. If the simulation is not successful a different arithmetic unit configuration should be sought, which will translate into a larger coefficient word length implementation. In the unlikely case that the system's response is still inadequate (for a selected arithmetic unit configuration), the sampling frequency should be reviewed and decreased. It is also worth checking the scaling of the original model. Sometimes, analogue controllers may include a chain of very big gains that can propagate quantisation errors.

5. Finite Word Length (FWL) arithmetic validation. At this stage, the control law, real-time equations and coefficients' quantisation have been validated.
Now the last step is to validate the actual hardware's arithmetic by replacing the rCSP's *soft model* by the rCSP's *hard model* and simulate the control system again. This model mimics the actual hardware arithmetic processing and therefore will spot any possible FWL related effect. This simulation is slower but accurate. Usually output mismatches at this stage are due to controller scaling problems and/or state variables' inadequate word length. These parameters can be easily changed in the software tools and the system immediately re-simulated.

![Figure 7.1: Proposed design-to-implementation methodology](image)

Once the FWL arithmetic has been validated, the system is ready to be pro-
grammed. If a stand alone controller implementation is selected, the software tool will generate a synthesisable boot module (in VHDL) with the required logic to interface the rCSP processor, transfer the control law and coefficients and start the control law execution. If a SoC solution is selected, the appropriate ANSI-C libraries are generated to interface and programme the different rCSP's modules. Those libraries are portable between different platforms so the design is not restricted to a particular CPU model or vendor. Once the libraries are generated, they just need to be compiled and linked with the control templates provided with the SoC solution (see Chapter 6).

### 7.4 rCSP Development Suite

This section describes the different modules that comprise the rCSP Development Suite. Details regarding its installation and launching can be found in Appendix D.3.

#### 7.4.1 Main Application Window

The main application window gathers under a common graphical user interface a variety of rCSP software modules such as cross-compiler (rcspcc.m), memory and coefficients' explorer, assembler and machine code viewer, arithmetic unit editor and a log window. By means of this tool a MIMO control algorithm can be described, discretised (if required), its real-time equations extracted and cross-compiled, and the required embedded software (and/or synthesisable boot logic) generated. Figure 7.2 presents the main application's window divided into five different sections that will be briefly described below. The practical examples provided in Sections 7.6 and 7.7 will further clarify the use and purpose of the different modules of this tool.

(a) The Menu Bar

The different menus are presented unfolded in Figure 7.3. The File menu is used to create, load or save a project. It also allows the user to change the project description parameters, to load and save different controller descriptions as well as finishing the
tool execution. The View menu allows to select which information should be shown on the screen (i.e. assembly code, machine code, memory structure) and enables or disables console messages and the log window. The Discretising Method menu allows the user to choose the discretisation method that will be used for generating the system's coefficients. The methods implemented are a subset of MATLAB's supported ones in its \texttt{c2dm} function, namely: zero-order-hold, bilinear transform (or Tustin) and matched pole-zero method. The Block Type menu allows the user to specify the general kind of system used in the block diagram description. The supported system types are: Continuous system (Laplace transform), Discrete Z-transform and Discrete Delta-transform. Mixed-type system description is possible as well as state-space representations. The Hardware Configuration menu is in charge of setting up certain processor parameters related with its arithmetic unit and the code generation. Table 7.1 describes the different items contained in this menu.

![Menu Bar unfolded](image)
**Table 7.1: Hardware Configuration menu items**

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always Adjust Code Word Length</td>
<td>If checked, the generated code word length will be adjusted even if the resultant size is &lt; 32 bits.</td>
</tr>
<tr>
<td>Adjust Frequency (NOP n)</td>
<td>If checked, the generated code will include wait cycles in the form of a NOP n instruction to adjust the control law's execution speed to the rCSP's clock.</td>
</tr>
<tr>
<td>Generate Coefficients</td>
<td>If unchecked, when the Generate Software Button is pressed, the real-time equations, memory structure and instruction's code are generated, but not the actual coefficients' value.</td>
</tr>
<tr>
<td>View/Modify AU Configuration</td>
<td>It opens the arithmetic unit configuration window (see Figure 7.4)</td>
</tr>
<tr>
<td>Saturated Arithmetic</td>
<td>If checked, the arithmetic unit will use saturated arithmetic.</td>
</tr>
<tr>
<td>Reset AU Configuration</td>
<td>It resets the arithmetic unit to its standards configuration.</td>
</tr>
</tbody>
</table>

Figure 7.4: Arithmetic unit configuration window

Once the system is simulated and the results are satisfactory, the *Implementation* menu allows the user to select the type of controller implementation required for the present application. If the *Embedded System Programming* menu-item is selected, the tool will automatically generate the required software libraries (in ANSI-C) that allow the embedded CPU to programme, interface and control the rCSP. These libraries will be linked with the rest of the CPU embedded software and will provide a full SoC embedded software solution. If there is no embedded CPU to control the rCSP, then the *VHDL Boot Module Generation* menu-item should be selected. In that case, the tool will automatically generate a synthesisable (VHDL) boot module that will transfer the control law and coefficients to the rCSP, starting its control execution afterwards. The *About* menu provides information about the author and some basic description of the software suite.
(b) Implementation Parameters

This area allows to define the sample frequency, the number of inputs and targets as well as the actual frequency (in MHz) that the rCSP will be clocked at.

(c) Control System Description Area

Control systems are often composed of a number of subsystems that are interconnected to form block diagrams of larger systems. A subsystem can be represented by a block with an input, an output and a transfer function (SISO). For instance, the system shown in Figure 7.6 is made of a number of these type of subsystems. Another type of subsystem can be a MIMO subsystem, represented by its state-space equations. Figure 7.5 provides a Simulink representation of such subsystems. In order to implement a control system into the rCSP, its cross-compiler must be provided with the real-time control equations. In order to facilitate this task, the control system can be easily described in the Control System Description area using a block diagram like representation. Then, the software tool will automatically generate the real-time equations based on this description.

![Figure 7.5: MATLAB's representation of a MIMO subsystem](image)

The notation accepted by the Control System Description area to describe SISO subsystems is presented next:

1. \( \text{Input} \times S_{\text{index}}(\text{order}) = \text{Output} \). This represents a single block whose output is equal to the product of its transfer function times its input. The index (integer) provides a way to enumerate and differentiate each block. The parameter order (integer) refers to the order of the systems.

2. \( \text{Signal}_1 + \text{Signal}_2 + ... + \text{Signal}_n - \text{Signal}_{n+1} - ... - \text{Signal}_m = \text{Output} \). This represents a summing junction of \( m \) signals, where the value of \( n \) of them is
added and value of the rest of them is subtracted.

3. Input * INT_index(gain identifier) = Output. This notation allows to describe a proportional integrator (PI).

4. Input * GAIN_index(gain identifier) = Output. This expression is used to represent gains in the system. The gain identifier is a string that would be used by the cross-compiler as symbol to refer to this particular coefficient.

In order to include discrete MIMO subsystems in the Control System Description area, the following notation should be used:

1. [Input1, Input2, ...] * MIMO_index(identifier) = [Output1, Output2, ...]
   General discrete MIMO subsystem as described by Equations (2.13) and (2.14) in Chapter 2. The discretisation method is not specified, hence it can implement any type of formulation.

2. [Input1, Input2, ...] * ZMIMO2DEL_index(identifier) = [Output1, Output2, ...]
   This command allows the user to generate automatically delta-based implementations of subsystems generated with the Z-Transform.

Comments can be introduced by starting a line with either the # or - character. The notation previously shown allows any LTI system to be described in a very intuitive and effective way. In order to illustrate the description of a general system using this notation, the block diagram represented in Figure 7.6 will be used as an example. This system has the following properties: U1, U2 and U3 are inputs; S1 is a third order filter, S2 is a proportional integrator with gain Ki and S3 is a second order filter; S4 is a first order filter and the output V is preamplified by a gain K.

This system can be readily described in the Control System Description area as:

```
# System Description Example
# --------------------------
IN1*ST(3)·y1
IN2*INT1(K1)·y2
IN3*ST(2)·y3
y1+y2-y3·u4
u4*ST(1)·y4
y4*GAIN(K)=Out1
```
Figure 7.6: Block diagram representation of a 3 input 1 output MIMO system

Once the block diagram has been defined in the Control System Description area, the next step is to provide the software tool with the internal description of each subsystem. This is done by tailoring the `genControllerCff.m` template for the particular application, including it afterwards within the project's search path (defined in the project parameters menu). Figure 7.7 shows an extract of this template. In order to define the integrator \( \text{INT1}(K_i) \) from Figure 7.6, only the template's line 56 needs to be modified, replacing \( K_{\text{int}\_\text{gain}} \) with \( K_i \)'s value. In the same example, the gain \( K \) should be provided in the template's line 49, replacing \( K_{\text{my}\_\text{gain}} \). In the same way, the subsystem \( S_1 \) can be defined in the template's line 40, replacing numerator and...
denominator by the actual values of this particular transfer function. Subsystems of higher index can be added just by creating a new case with the same structure (i.e. copy-paste from the template's code) and changing the indexing. The indexing is represented on every case statements case (kxx_IDX+n) as n. All these concepts will be clarified in the examples included at the end of this chapter.

(d) Processor Selection Drop List and Buttons

Multi-rCSP control applications can be developed using the MIMO Control System Development application. The Multiprocessor List button opens a window where the list of rCSP processors used in the current project are shown. This window allows the user to add, delete and rename the processors in that list. The rCSP Processor drop list is used to select/activate a processor from the list. The controller description and selected implementation parameters act only on the selected processor. The latest changes can be made available to the active processor by pressing the Apply button.

Once a valid control description is provided to the processor, together with the implementation parameters, the Generate Software button starts the compilation process. First, the real-time equations are extracted from the controller description. Then, these equations are sent to the rCSP's cross-compiler, which generates the coefficients, the rCSP's software and memory structure.

(e) Real-Time Equations Area

After compilation, the generated real-time equations are shown in this area. In order to give full control over the generation process, the real-time equations can be edited and assembler code can be also included. However this feature is included for experimental development and should not be needed in general.

7.4.2 Memory Structure Module

The Memory Structure module provides not only a means of displaying the resultant rCSP memory structure, but also a link to the simulation modules. This interface is
Chapter 7: Design Methodology

divided into four different areas (see Figure 7.8(a)) described in Table 7.2.

Figure 7.8: Memory Structure general window and coefficients’ value display

<table>
<thead>
<tr>
<th>Area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor’s List</td>
<td>The information displayed in this window is related to the selected processor</td>
</tr>
<tr>
<td>State Variables</td>
<td>The memory structure of the state variables and inputs is here presented.</td>
</tr>
<tr>
<td></td>
<td>From this list, the user should choose the controller’s outputs</td>
</tr>
<tr>
<td>Coefficients</td>
<td>The coefficients’ memory structure is shown here.</td>
</tr>
<tr>
<td></td>
<td>Right-click button on the list to open a context menu that allows to display</td>
</tr>
<tr>
<td></td>
<td>the coefficients’ values and their quantisation error (see Figure 7.8(b))</td>
</tr>
<tr>
<td></td>
<td>on a separate window.</td>
</tr>
<tr>
<td>Export Button</td>
<td>This button updates the rCSP’s assembly and machine code including the</td>
</tr>
<tr>
<td></td>
<td>selected write instructions and makes this changes available for simulation.</td>
</tr>
</tbody>
</table>

Table 7.2: Memory structure module interface description

7.4.3 Simulink’s rCSP Models

In the last few years, Simulink has become the most widely used software package in academia and industry for modeling and simulating dynamic systems. Simulink provides instant access to MATLAB’s extensive variety of resources, and therefore to the functionalities implemented in the rCSP Development Suite.

In order to integrate the rCSP processor within Simulink, two different models of the processor have been implemented and made available as a Simulink library (rCSPv30_LIB.mdl). These models (see Figure 7.9) differ in only one aspect, namely
the modeling of the arithmetic unit used during the simulation. Each model possesses two input ports (i.e. coefficient set and input signals) and two output ports (i.e. output data and number of algorithm's cycles executed). This library also includes three types of coefficient block. The first one (red) contains the controller coefficients represented with MATLAB's full precision (i.e. 32-bit/64-bit floating-point precision). In the second one (orange), the coefficients are rounded to the actual precision of the selected arithmetic unit configuration, and stored in decimal format. The third one (grey) contains the actual binary representation use by the actual hardware of the rounded coefficients.

![Simulink's rCSP models and coefficients modules](image)

Figure 7.9: Simulink's rCSP models and coefficients modules

The characteristics and use of the two rCSP models are described next:

- **Software-based arithmetic unit model (soft model).** In this model, the arithmetic operations are simulated using MATLAB's standard 32-bit floating-point arithmetic. No binary strings are used either to represent the coefficients, or the state variables, or the algorithm code, hence obtaining a fast simulation. This model, when used in combination with the red coefficients block, allows the user to test the quality of the algorithm disregarding the numerical errors that the finite word length arithmetic used by the rCSP may introduce. If the model is used in combination with the rounded coefficients block, then it allows the user to analyse the effects that the coefficient's quantisation alone has over the system's response.

- **Hardware-based arithmetic unit model (hard model).** When this model is used, all the variables, coefficients and code are treated as binary strings. The arithmetic operations are performed using the rCSP's hybrid arithmetic and specific
word length. This model behaves numerically the same as the real hardware implementation and therefore it is of primary importance for the validation of the design. Due to its internal modeling of the arithmetic unit, all the numerical issues related to the use of finite word length arithmetic (for coefficients and state variables representation) will be detected (e.g. overflow, sign change, quantisation, etc...).

7.5 Design Flow

The design and implementation concepts previously described in this chapter can be summarised in the design flow shown in Figure 7.10. In this figure, the partitioning of tasks between the control engineer and the \textit{rCSP Development Suite} is clarified.

As can be seen, the design-to-implementation methodology presented in this thesis allows control engineers to focus on their field of expertise (controller design), leaving the high-performance embedded implementation issues to the software tool. The designer provides the tools with a description of the controller (for example in the s-plane) and a set of implementation parameters (e.g. sampling frequency, arithmetic unit configuration, etc.), and it is the tool's duty to generate a discrete model of the system (based on the modified canonic $\delta$ form, for better real-time performances), to obtain the real-time control equations, to programme the rCSP, and to generate the memory structure and the control coefficients. In order to do so, the tool integrates
with and automatically operates the rCSP-cross-compiler, which transforms the real-time control equations into rCSP assembly and machine code, providing a memory description map, hardware utilisation and quantisation figures.

The simulation of the system is linked with the compilation results and therefore it can be automatically started. Since such a simulation is based on the Simulink toolbox, which is the de facto industry standard tool-set in the aerospace and automotive fields among others, there is no need to port the existing system model and control environment into another tool, thus saving extra time and effort. The simulation can be done at different levels, providing a highly intuitive way of testing and tuning the system just by changing parameters, without the need of any system's coding.

Once the control system is simulated and tuned, traditional development methods would require some code to be developed to boot and initialise the system, and to bring it to its initial working point. As shown in the design flow presented in Figure 7.10, this is part of the software tool's tasks, and therefore it is automated. In the case of a system-on-chip application, the tool generates ANSI-C libraries capable of booting the rCSP, adapting its coefficients and interfacing its different memory-mapped modules. These libraries are architecture independent and therefore portable between different embedded CPU architectures. Once these libraries are generated, they just need to be compiled and linked with the control templates provided with the SoC solution. These templates are real examples of different adaptive techniques, data filtering routines and user interaction schemes ready to be used. In case the rCSP is used without an embedded CPU, the tool generates the required synthesisable logic (VHDL) to boot the rCSP and start the execution of its algorithm.
7.6 Design Flow Example (I): F14 Autopilot Flight Controller

This example provides a step-by-step guide to the design and implementation of a digital embedded controller based on the rCSP processor. This example will:

1. Exemplify the utilisation of the rCSP Development Suite.

2. Demonstrate the description of a control system in the software tool. In this particular example an analogue controller is used as source to programme the rCSP.

3. Clarify the different steps needed in the design an implementation of an embedded digital controller, demonstrating the automation of the steps described in the thesis.

4. Demonstrate the numeric robustness of delta formulated controllers that allows their coefficients to be implemented with a minimal amount of bits.

7.6.1 System Overview

The goal is to design and simulate an aerospace control application: a digital flight control system for the longitudinal motion of a Grumman Aerospace F-14. This system is adapted from a MATLAB example provided as a Simulink demonstrator for aerospace applications.

The basic elements that comprise this system are:

- **Pilot.** On a fly-by-wire system, the pilot actuates on the commanding stick setting the desired trajectory of the plane. In this particular example, the pilot commands the longitudinal movement of the plane (i.e. angle of attack). In the simulation, the stick can be manually commanded or set to perform a cyclic path.

- **Wind Gust Model.** This block models the Wind Gust using a Dryden model.
• **Plane model.** In this example, the plane is modeled as a 3 input - 2 output MIMO system.

• **Autopilot system.** The Autopilot system acts as a servomechanism trying to make the plane to follow the target set by the pilot. It is originally designed as 3 input - 1 output analogue controller.

### 7.6.2 Analogue autopilot subsystem

The original analogue design of the controller is shown in Figure 7.12. In this example, this controller will be digitally implemented in the rCSP.

![Analogue Controller Block Diagram](image_url)
7.6.3 Digital autopilot formulation

Describing the controller

Once the new project is created (see Figure 7.13), the next step is to generate a digital version of the analogue autopilot.

Following the syntax presented in Section 7.4.1 the description of the analogue controller's block diagram (see Figure 7.12) is provided into the Control System Description area, as shown below:

```plaintext
 Stick*S1(1)=y1 
 Alpha*S2(1)=y2 
 q=S3(1)=y3 
 y2=GAIN2(Ka)=y2g 
 y3=GAIN3(Kp)=y3g 
 y1-y2-g=y_int 
 u_int*INT1(KI)=y_int 
 u_int=GAIN4(KI)=yg 
 y_int+yg=Elevator
```

In order to fully describe the control system an internal description of each subsystem should be provided by tailoring the `genControllerCff.m` template (see Figure 7.14). As can be seen, the description of each element requires only a modification in
a small section (highlighted) of a single line of code. Once the autopilot block diagram is fully described, the main implementation parameters are introduced as follows:

- **Number of Inputs**: 3
- **Desired Sampling Frequency**: 100 Hz
- **Discretisation Method** (Menu): Zero-order-hold.
- **Block Diagram Type** (Menu). The original controller is described in terms of transfer functions in s so Analogue (Laplace) is selected.
- **Hardware Configuration** (Menu): Generate Coefficients (√), Saturated Arithmetic (√). The standard rCSP arithmetic unit configuration is selected by default.

At this point, the controller's internal and external description has been provided and all the design and implementation parameters are set, so the tool is ready to generate the digital controller as soon as the Generate Software button is pressed. This
action instructs the rCSP Development Suite to generate the appropriate state-space canonic delta subsystems and to extract its real-time equations. Then it executes the rCSP's cross-compiler which transforms these equations into rCSP's assembly and machine code (see Figure 7.15(a) and 7.15(b)). Then, the coefficients are calculated for the selected arithmetic unit configuration and the memory structure is generated and presented (see Figure 7.15(c)). Figure 7.16 shows the rCSP Development Suite's main module with the generated real-time control equations.

Figure 7.15: Generated assembly code, machine code and memory structure

7.6.4 Simulation

The original Simulink model used to test the analogue control system is re-used, substituting the analogue controller with a Simulink model of the rCSP. This model can be found in the rCSP's Simulink library accessible from the Application Launcher. Figure 7.17 shows the Simulink model of the F-14 flight control system with the rCSP processor as controller.
Validating Real-Time Equations and Sampling Frequency

Following the methodology described in Section 7.3 the soft model of the rCSP processor (rCSP v3.0 Soft - 1) and the MATLAB's floating-point coefficient block are initially used. This allows the user to validate the selected sample time ($T_s$) and the generated real-time equations. After generating the real-time equations and calculating the coefficients, the desired controller outputs should be selected and exported. This is done in the Memory Structure window as shown in Figure 7.18. In this example, the following variables are selected as outputs: $y_1$, $y_2$, $y_3$ and Elevator. The first three are chosen for monitoring purposes while the last one is the actual signal that commands the wings' actuators. Then, the system is simulated (simulation time = 30 seconds) and its response analysed. Figure 7.19 presents a comparison between the Pilot's command (desired angle of attack), the response obtained by the analogue controller and the one obtained by the rCSP. As can be seen in this figure, the dynamic response of the plane using the rCSP fulfils perfectly the design requirements, hence validating the generated real-time equations. The sample rate is also valid when using MATLAB's floating-point precision, but its final validation will depend on the selected arithmetic unit configuration.
After validating the system’s response provided by the real-time equations executed at the chosen sample rates, the next step is to validate the coefficient quantisation for the selected (standard) arithmetic unit configuration (i.e. 6-bit mantissa, 5-bit
Figure 7.18: rCSP outputs selection

Figure 7.19: Airplane’s response to the pilot’s commands using the original analogue controller and the soft model of the rCSP with MATLAB’s floating-point arithmetic exponent and bias=30). In order to do so, the rCSP should receive the rounded off version of the coefficients provided in the rCSP’s Simulink Model Library. The new response of the plane is presented in Figure 7.20. As can be seen, the airplane’s dynamic response with quantised coefficients (standard arithmetic unit configuration) has a slightly smaller overshoot and its position error differs in less than 1% from the one obtained by the un-quantised real-time equations.

Figure 7.21 compares the system’s response for different coefficient models, while Table 7.3 summarises the absolute position-error obtained by these models. The position-error is measured with respect to the command signal and to the system’s response obtained by the original (analogue) controller. These results demonstrate the numerical robustness of the δ formulation, even for a minimal coefficient word length.
Chapter 7: Design Methodology

Figure 7.20: Airplane's response using rCSP's soft model with rounded off coefficients

Figure 7.21: Airplane's response for a variety of coefficient configurations

<table>
<thead>
<tr>
<th>Reference</th>
<th>Coefficient's Mantissa Length (bits)</th>
<th>Analogue controller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>m=2</td>
<td>m=3</td>
</tr>
<tr>
<td>Pilot's Command</td>
<td>6.5%</td>
<td>7.0%</td>
</tr>
<tr>
<td>Analogue controller</td>
<td>8.0%</td>
<td>5.5%</td>
</tr>
</tbody>
</table>

Table 7.3: Absolute (peak) position-error for a number of coefficient configurations

Validating the Arithmetic Unit's Finite Word Length Architecture

Once the coefficient quantisation has been validated for the selected word length, the next step is to test the hardware model of the rCSP. This is done by substituting
the current (soft) model or the rCSP for the hard one that can be found in the rCSP's Simulink Model Library, and replacing the current coefficients block with the one that contains the binary representation of the coefficients (Hard CFF block). The system is simulated for a subset of the previously tested arithmetic unit configurations. As Figure 7.22 shows, the hardware simulation results match the predictions of the previous validation stage, and therefore, the controller's implementation can proceed further.

Figure 7.22: Command vs Actual Angle of Attack using the actual rCSP's hardware model

In summary, the rCSP Development Suite provides the control engineer with different results for different configurations so he/she can decide which arithmetic unit configuration to take depending on the specifications and resource constraints inherent to the project.

7.6.5 System Programming

Once the control system has been successfully simulated using the rCSP's hard model, the controller is ready to be implemented. The first step now is to choose between a system-on-chip implementation or a stand alone one. In this example the first one is featured. When a controller is part of a SoC design governed by an embedded CPU, it is necessary to write a significant amount of code to configure, control and communicate with the different devices and peripherals that comprise the system-on-chip design. This thesis provides working templates of rCSP-based SoC real-time control solutions, which as explained in Chapter 6 are simple to adapt to the particularities
of each project. The embedded CPU code required for booting, programming and
commanding the rCSP processor is automatically generated by the software tool.
Figure 7.23 shows an extract of one of the generated libraries (rcspv30_boot.c). The
function Boot_Algorithm(), whose code is presented in this figure is in charge of
transferring the compiled control algorithm into the rCSP’s CL-MOD.

```c
/* Function : Boot_Algorithm */
/* Called by : main.c */
/* Description: Subroutine that programs the CL-MOD */
/* with the desired control law. */

void Boot_Algorithm(void)
{
  volatile unsigned int *CL_MOD;
  volatile unsigned int *IL;
  static unsigned int PROG_SIZE = 21;
  static unsigned int INST[] = {
    0x10001020, /* 2_00010000100000000000001000000000; MUL y1,c1_1,x3_1 */
    0x00100100, /* 0_00000000000000000000000000000000; MAC y1,c1_1,x1_1 */
    0x00000100, /* 0_00000000000000000000000000000001; MAC x1_1,b1_1,x3_1 */
    0x10001020, /* 2_00010000100000000000001000000000; MUL y1,c1_1,x3_1 */
    0x00100100, /* 0_00000000000000000000000000000000; MAC y1,c1_1,x1_1 */
    0x00000100, /* 0_00000000000000000000000000000001; MAC x1_1,b1_1,x3_1 */
    0x10001020, /* 2_00010000100000000000001000000000; MUL y1,c1_1,x3_1 */
    0x00100100, /* 0_00000000000000000000000000000000; MAC y1,c1_1,x1_1 */
    0x00000100, /* 0_00000000000000000000000000000001; MAC x1_1,b1_1,x3_1 */
    0x10001020, /* 2_00010000100000000000001000000000; MUL y1,c1_1,x3_1 */
    0x00100100, /* 0_00000000000000000000000000000000; MAC y1,c1_1,x1_1 */
    0x00000100, /* 0_00000000000000000000000000000001; MAC x1_1,b1_1,x3_1 */
    0x10001020, /* 2_00010000100000000000001000000000; MUL y1,c1_1,x3_1 */
    0x00100100, /* 0_00000000000000000000000000000000; MAC y1,c1_1,x1_1 */
    0x00000100, /* 0_00000000000000000000000000000001; MAC x1_1,b1_1,x3_1 */
  }; Setting the End PC value into the rCSP v3.0 REGISTERS
  SOC_RCSPV3_REGS(REG_END_PC) = PROG_SIZE-1;

  /* Downloading the Control Law into the rCSP v3.0 CL-MOD */
  CL_MOD = (volatile unsigned int*) SOC_RCSPV3_CL_MOD_BASE;
  IL = (volatile unsigned int*) SOC_RCSPV3_IL_BASE;
  console(DEBUG_CODE, "----- Sending PROGRAMMING Command -----\n
  for (int i=0; i<PROG_SIZE; i++)
    "((CL_MOD++)=INST[i]);/* CL-MOD address+32 bits */

  console(DEBUG_CODE, "----- Sending END_PROGRAMMING Command -----\n
  return;
} /* end Boot_Algorithm */
```

Figure 7.23: Extract from the module rcspv30.boot.c
7.7 Design Example (II): Hard Disk Drive (HDD) Controller with Kalman Filtering

7.7.1 Example Objectives

In the previous design example an analogue controller was directly transferred into the rCSP processor and operated at 0.1 kHz. The goal of this second example is twofold. First, it shows how to implement a system which is described by mixing digital and analogue components. Second, the control system will be operated at 10 kHz (100 times faster than in the previous example). In summary, this example will:

1. Clarify the description of a mixed control system in the software tool. In this particular example an analogue controller is used in conjunction with a discrete state-space description of a Kalman filter.

2. Demonstrate the numerical benefits of δ-operator implementations (especially at high-sample rates) upon the traditional shift-operator formulation. In order to do so, the Kalman filter block will be implemented in both the modified delta and the \( z \) form.

3. Demonstrate the seamless tailoring of the rCSP processor for a particular application in order to obtain the required performance.

4. Show how an rCSP-based controller can address real-time control systems operated at very high sample rates (e.g. micro-electromechanical systems), without any changes in the design-to-implementation steps earlier described.

7.7.2 System's Overview

The goal is to design and implement an embedded micro-electromechanical system (MEMS) operated at high-sample rates: a Hard Disk Drive (HDD) head's position controller that incorporates Kalman filtering to obtain better performance in the presence of measurement noise.
The HDD's model and the controller (a phase lead compensator) are shown in Figure 7.24 (a) and (b) respectively. As can be seen, the HDD (and hence the Kalman filter) is third order and so the combined controller is order four. The system parameters are summarised in Table 7.4 and the complete control system is depicted in Figure 7.25. In this example, the HDD head’s position sensor is known to provide noisy measurements. In order to achieve better performance, the controller will receive a statistical estimation of the head’s position provided by a Kalman filter stage. This filter is calculated using MATLAB’s function \texttt{kalman(Plant, Q, R)} (\textit{Control System Toolbox}), which returns the Z-Transform-based state matrices of the filter.

![Figure 7.24: Block diagram description of (a) HDD's model (b) Controller](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inertia of arm and head</td>
<td>$J_m$</td>
<td>$3 \cdot 10^{-6} N \cdot m \cdot s^2 \cdot /rad$</td>
</tr>
<tr>
<td>Friction</td>
<td>$b$</td>
<td>$10^{-3} \text{kg/m/s}$</td>
</tr>
<tr>
<td>Armature resistance</td>
<td>$R$</td>
<td>$6 \Omega$</td>
</tr>
<tr>
<td>Motor constant</td>
<td>$K_m$</td>
<td>$9 \cdot 10^{-2} N \cdot m/A$</td>
</tr>
<tr>
<td>Armature inductance</td>
<td>$L$</td>
<td>$0.5 \text{mH}$</td>
</tr>
<tr>
<td>Sensor gain</td>
<td>$K$</td>
<td>$326$</td>
</tr>
<tr>
<td>Actuator gain</td>
<td>$K_A$</td>
<td>$0.0046$</td>
</tr>
<tr>
<td>Output gain</td>
<td>$K_O$</td>
<td>$0.67$</td>
</tr>
<tr>
<td>Model Noise Covariance</td>
<td>$Q$</td>
<td>$1$</td>
</tr>
<tr>
<td>Measurement Noise Covariance</td>
<td>$R$</td>
<td>$0.15$</td>
</tr>
</tbody>
</table>

Table 7.4: System Parameters. HDD’s values except for $b$ are courtesy of \textit{Hitachi Corporation, Japan}
7.7.3 Digital control system formulation

Control system description

After creating a new project the sampling frequency is selected and control system’s block diagram (see Figure 7.25) is first outlined at a high-level in the Control System Description area (see Figure 7.26). In this example, the Kalman filter is alternatively defined using the two MIMO-description functions presented in Section 7.4:

1. \([\text{action}, y_v] \cdot \text{ZMIMO2DELn(identifier)} = [y_{est}]\)
2. \([\text{action}, y_v] \cdot \text{MIMOn(identifier)} = [y_{est}]\)

where the first parameters between brackets are the inputs to the MIMO subsystem while the parameters after the equal symbol are the outputs. In this case, the inputs are the action calculated by the controller, which is defined by the subsystem \(S1\) (the controller), and the measured value of the head’s position provided by the sensor. The output parameter of the system is the estimated value of the head’s position. Notice that \(\text{ZMIMO2DELn}\) transforms automatically a \(z\)-based MIMO state-space representation into its equivalent modified delta from, while \(\text{MIMOn}\) compiles without modification a given state-space representation. Therefore, unless the \(\text{ZMIMO2DELn}\) syntax is used, the controller’s implementation will inherit the numerical problems typical to the \(z\)-based formulation.
After providing the block diagram description of the control system (see Figure 7.26), the template file genControllerCff.m should be tailored to provide the internal description of each subsystem. Figure 7.27 presents a snapshot of the code area where the different blocks are described, highlighting the modifications to the original template. Notice that the comments on the template's code advise which areas should be edited. In order to define the discrete Kalman MIMO system, the template requires the values of the state-space matrices \((A,B,C,D)\), which are calculated in the function \([A,B,C,D]=generate_kalman_matrices(Ts)\).

Figure 7.28 shows a snapshot of the template file genControllerCff.m tailored to use the original Z-Transform-based Kalman filter, hence with the MIMO syntax.

### 7.7.4 Simulation

Figure 7.29 presents the Simulink model used to simulate the control system. In order to validate the system's response and to appreciate the effect of the head position's
estimation in the case of noisy measurements, the model is divided into three different areas. The first one (in green) is in charge of generating a Gaussian noise that (if activated) will be added to the actual HDD head’s position measurement. The second one (in blue) contains the rCSP-based HDD’s control system, which uses an estimation of the head’s position provided by the Kalman filter stage. The third one (in pink) contains the HDD’s control system (without filtering) based on the original analogue controller which directly interfaces the HDD. Following, the delta-formulated system will be analysed, comparing the results at the end of each validation stage with the ones obtained by the z-based formulation.
Validating Real-time Equations and Sample Rate

From the Memory Management Window the following controller outputs are selected (in order) and exported: action (used to position the HDD's head), y_est, error and yv. In order to validate the controller’s requirements in terms of position error, overshooting and settling time, the control system is first simulated (simulation time= 0.1 seconds) without applying any measurement noise. The system’s response due to the analogue controller and rCSP running the real-time equations is presented in Fig-

Figure 7.28: genControllerCff.m tailoring (MINOn syntax)
Validating the Coefficient Quantisation

After validating the real-time equations, the coefficient quantisation should be analysed. First, the delta-formulated system is simulated. Figure 7.31 presents the system's response and position-error percentage (in absence of measurement noise) for a number of rCSP's arithmetic configurations (ml stands for mantissa length). The system's response and the actions generated by the analogue and digital controllers in the presence of Gaussian measurement noise are compared in Figure 7.32. As can be readily seen, the effects of the measurement noise are largely reduced due to the output's estimation performed by the Kalman filter stage. This becomes even more evident when analysing the actions generated by both controllers, as shown in Figure

Figure 7.29: Simulink test bench of the HDD control system

ure 7.30(a), observing a similar behaviour. However, in the presence of measurement noise (Figure 7.30(b)), the Kalman filter stage implemented in the rCSP generates a smoother response.
Chapter 7: Design Methodology

Figure 7.30: System’s response comparison for validating the real-time equations

Figure 7.31: System’s response with quantised (delta) coefficients

7.32(b). In an attempt to compensate for the measurement noise, the action generated by the analogue controller fluctuates constantly within a wide range at the same rate as the sampling one. On the contrary, the action generated by the rCSP is much smoother and its final fluctuation range is largely decreased. These results confirm
Chapter 7: Design Methodology

Figure 7.32: Analogue and delta-formulated digital controller \((m_l = 6)\) comparison in presence of measurement noise

the validity of the filter itself together with the quantisation of its coefficients.

Figure 7.33: System’s response with quantised \((z\text{-based})\) coefficients

As Figures 7.31 shows, when the system is fully implemented in the delta-form, the \(r\text{CSP}'s\) standard configuration (i.e. \(m_l = 6\)) provides sufficient coefficient accuracy to obtain a position-error smaller than 0.5%.

The simulation's results of the control system implementing a \(Z\text{-Transform-based}\)
Kalman filter are presented in Figure 7.33. As can be seen, in order to obtain a similar position-error, the shift-operator formulation requires a mantissa length of 16 bits, which translates into doubling the coefficient word length with respect to the delta-formulation. This result exemplifies the typical shift-operator's numerical problems at high sample rates.

Validating the FWL arithmetic

After validating the coefficient quantisation, the complete finite word length arithmetic of the control system should be analysed by simulating the rCSP using its hard model. The results of the system's simulation, without and with Gaussian measurement noise, are presented in Figure 7.34 and 7.35 respectively. It can be appreciated in Figure 7.35(b) that the action generated by the analogue controller is highly conditioned by the measurement noise, while the one generated by the rCSP is very much smoother due to the HDD's head's position estimation. As can be seen, the standard arithmetic unit configuration of the rCSP provides enough numerical precision to successfully address real-time high-performance delta-formulated control systems operated at very high-sample rates.

Figure 7.34: System's response with the rCSP's actual hardware model (noiseless)
Figure 7.35: System's response with the rCSP's actual hardware model (including Gaussian measurement noise)

Since the rCSP model used in this simulation mimics the actual processor's arithmetic unit's hardware implementation, the results obtained provide the most accurate prediction of the real control system's behaviour. Therefore, in view of these results the control system (in its delta formulation) is validated and ready to be programmed.

As a final comparison, the z-based controller is simulated and briefly analysed next. In order to achieve the same level of performance using the original z-based version of the Kalman filter generated by the MATLAB's Control System Toolbox, the state variable's configuration should be modified. The reason is that the higher level of coefficient accuracy (16-bits mantissa word length) obtained in the previous validation stage also requires increases in the state variables' word length, which in the standard rCSP's arithmetic unit configuration only provide a resolution of $2^{-12} \approx 2.4 \cdot 10^{-4}$. Figure 7.36 shows the system's response for a number of state variable's configurations. As can be seen, the variables need at least 20 fractional bits to obtain a position error somehow similar to the one achieved by the delta form representation with the standard rCSP configuration.

In summary, the delta-based implementation of the control system requires much shorter word lengths to represent its coefficients and state variables than the z-based counterpart (see Table 7.5).
Chapter 7: Design Methodology

Figure 7.36: System’s response (z formulation) with the rCSP’s actual hardware model (noiseless)

<table>
<thead>
<tr>
<th>Formulation</th>
<th>Coefficients</th>
<th>State Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>word length</td>
<td>mantissa exponent</td>
</tr>
<tr>
<td>Delta-Transform</td>
<td>11 6 5</td>
<td>27 12 12 3</td>
</tr>
<tr>
<td>Z-Transform</td>
<td>21 16 5</td>
<td>35 12 20 3</td>
</tr>
</tbody>
</table>

Table 7.5: Implementations’ arithmetic word length requirements (in bits)

7.7.5 System Programming

In this example a stand alone implementation is featured and therefore an extra module should be provided capable of communicating with the rCSP processor via its APB interface, transferring the control algorithm and the coefficients, and bringing it into control. This module is automatically generated by the rCSP Development Suite by selecting “Generate VHDL PROM module” from the Implementation menu. Figure 7.37 shows an extract the module generated, where the control law is encoded into rCSP’s machine code and placed within a ROM-like structure.
Chapter 7: Design Methodology

7.8 Future enhancement

One of the main objectives of this thesis is the development of a design flow that allows control engineers to focus on their field of expertise, relieving them, for instance, from complex hardware-dependent development issues. The rCSP Development Suite is designed for this purpose, allowing the user to configure, simulate and programme rCSP-based controllers directly from control block diagrams. However, once the processor is configured, its design needs to be synthesised, obtaining in the case of an FPGA implementation a new bit-stream that has to be downloaded into the FPGA device. This process requires the use of synthesis and place & route tools and their (usually expensive) licensing.
Chapter 7: Design Methodology

In order to take full advantage of the rCSP processor without the need of these tools, the following scheme is proposed as a future enhancement. As Figure 7.38 shows, the scheme is based on a client-server architecture that communicates over the world-wide-web (WWW). Once the control algorithm is designed, the rCSP Development Suite (with the appropriate plug-in) connects (e.g. via HTTP) with an rCSP-server and sends it the rCSP's configuration parameters (e.g. mantissa size, number of SAR units, FPGA device, etc.) required to implement the control algorithm. The server side could consist of two dedicated desktop servers. One of them should be a bitstream repository of the most common rCSP configurations (for a variety of FPGA technologies), and therefore if the requested pair "configuration-FPGA" is available no further processing would be required. The other server should be in charge of generating the bitstreams for any configuration that is not stored in the repository. This server would need to have installed the appropriate (licensed) synthesis and place & route tools, the VHDL model of the rCSP processor and the rest of the system-on-chip architecture (if required). Once the bitstream is ready, it can be transferred (e.g. via FTP) to the client side, including the associated implementation figures. In that way, if the figures are satisfactory, the control engineer would only need to download the received bitstream into the FPGA device. Otherwise he/she could modify the selected parameters (e.g. new FPGA device, new rCSP configuration, etc.) and request a new bitstream.

Figure 7.38: Client-Server architecture for remote bitstream generation
7.9 Summary

This chapter presented the rCSP Development Suite, a MATLAB-based software tool developed to support the design of rCSP-based control solutions. This tool combines Simulink's renowned power with an accurate model of the rCSP's hardware architecture, which allows the modelling, analysis and simulation of complex real dynamics systems controlled by the rCSP. Furthermore, once the control systems is validated, the tool automatically generates the hardware and/or software implementation modules.

In order to demonstrate the design-flow, utilisation and benefits of the rCSP Development Suite, this chapter concluded presenting two detailed implantation examples: a digital flight control system for the longitudinal movement of a Grumman Aerospace F-14, and a hard-disk-drive controller that includes Kalman filtering. This last example also demonstrated quantitatively the superiority of the delta formulation at high-sample rates.
Chapter 8

Conclusions

8.1 Chapter objectives

This chapter concludes this thesis evaluating the results obtained. The objectives of this chapter are:

- To summarise and review the objectives of the thesis
- To summarise the outcomes and results of each chapter
- To analyse how well these results met the thesis objectives
- To discuss the limitations of the current research and to present ideas for a potential future research.

8.2 Summary of objectives

The objective of this research (as presented in Chapter 1) is to provide a unified solution to the development and implementation of real-time embedded controllers capable of delivering the performance and functionalities that modern advanced real-time control systems require.
8.3 Summary of the research

At the beginning of this work the research scope and objectives were clarified (Chapter 1) and an exhaustive literature review (Chapters 2 and 3) were performed. The literature review focused on three main aspects:

- Identification of current and future real-time embedded control design and implementation requirements
- Analysis of different technologies used for the implementation of current control systems
- Study of the architectural trends and evolution that modern processing devices are undertaking in order to cope with the functionalities and implementation targets that arise in modern embedded real-time control applications.

The first result obtained at this initial stage was the confirmation that a unified approach to the problem of designing and implementing modern real-time embedded controllers can clearly benefit designers and consumers. This result is based on how the development of new technologies (where often real-time embedded controllers play a key role) is driving control designers to the edge of their field of expertise, spending most of their time in complex high-performance hardware/software co-design issues. Furthermore, continuous developments in the semiconductor industry will certainly put an extra load on control engineers, which will be required to increase the functionalities and performance provided by their designs, shifting progressively their productive time towards architecture-dependent hardware/software co-design task.

After analysing the benefits of a unified control system design-to-implementation approach, a fit-to-purpose hardware architecture capable of meeting the performance and resource targets is investigated. The functionalities and performance identified during the literature review, which the proposed solution needed to address, were:

1. Real-time control operation at high sample rates. The controller should be numerically robust at such sampling frequencies.
2. Implementation size and power consumption awareness.


4. The controller architecture should be reliable, reusable, reconfigurable and upgradable in order to meet future requirements.

5. The controller architecture should be able to work as a stand alone device but also it should be easily integrated in complex hardware systems (such as SoC designs) as a soft IP core.

6. Hardware/software co-design simplification. Elimination of controller programming tasks and therefore controller debugging.

The numerical issues related with the use of the Z-transform at high sample rates were introduced in Chapter 2, while Chapter 4 provided an insight into the Delta-transform. This δ-operator based transform is widely acknowledged to be more numerically robust at high sample rates, and therefore much more suitable for real-time implementation than its shift-operator based counterpart. Chapter 4 also analysed the finite word length implementation issues typical to digital systems that operate with short word length floating-point representations.

After analysing the strengths and shortcomings of different devices (Chapter 3) commonly used in control implementations (DSPs, MCUs, and embedded microprocessors) together with their supporting tools (i.e. compilers) it was soon realised the potential benefits of using a targeted architecture versus a general off-the-shelf device. Such special-purpose architecture should be designed specifically to fulfill the previously identified requirements, but as explained in Chapter 5, avoiding overspecialisation.

The design and development of such an architecture went through several stages, concluding with the final reconfigurable soft IP control processor presented in Chapter 5, namely the Reconfigurable Control System Processor (rCSP).

As current control applications grow in complexity, there is a clear need for architectures that apart from implementing highly-specialised control law processing
elements, also include more general processing modules that provide higher-levels of abstraction and functionality. Therefore, this thesis developed a customisable multi-processor system-on-chip solution (presented in Chapter 6) capable of addressing complex adaptive real-time systems and seamlessly interfacing with analogue and digital devices.

In parallel with the development of the rCSP and the SoC architecture, the hardware/software co-design simplification idea was forming. At the beginning it started as a simple assembly language cross-compiler for the rCSP, but it was obvious that the possibility of generating machine code directly from control equations (rather than assembly code) would represent the real improvement. Then a system generator was created to produce SISO 5-operator based systems, and shortly after the first drafts of what would be the rCSP Development Suite started taking shape. The results of this part of the research were presented in Chapter 7. In this chapter, the rCSP Development Suite was described in detail together with its design flow. A design-to-implementation methodology based on the rCSP processor and its development tool was also presented and demonstrated in two detailed examples.

8.4 Summary of achievements

This research addressed the problem of complex high-performance real-time embedded control implementations operated at high sample rates. Its focus was threefold: mathematical formulation of the controller, optimised fit-to-purpose hardware architectures, and automated system programming. The original contributions of the thesis included:

- Design and development of a novel reconfigurable control system processor (rCSP) developed as a soft IP core that can be tailored for each particular application, optimising in that way its implementation and reducing its power consumption. Its fit-to-purpose design makes it especially suitable for addressing demanding real-time control systems.

- Design and development of a novel multi-processor system-on-chip architecture
targeted for advanced embedded real-time control solutions, which provides state of the art functionality and performance and can be implemented in low-cost FPGA devices.

- Design and development of a software suite (*rCSP Development Suite*) that supports and automates the design-to-implementation stages of *rCSP*-based control solutions. Its novel design flow and implementation methodology allows control engineers to focus on the control algorithm's development, while automating most of the hardware/software co-design aspects, hence reducing the implementation time by removing complex programming and hardware related issues, yet obtaining the highest performances.

### 8.5 Future Work

As usual, due to the finite time available in a PhD research and the wide scope of the fields addressed in this investigation, not all the research paths could be fully explored. For the same reason, some simple but time-consuming implementation possibilities could not be finalised, favouring research activities rather than mere routine coding. Some examples of these cases and possible extensions to the work are summarised next.

- Extension of the *Control System Description* area functionality to allow the definition of the controller by means of Simulink block diagrams.

- Extension of the simulator capabilities to enable the simulation of the embedded CPU's adaptation routines. This could be easily achieved by using MATLAB's Stateflow, which allows the simulation of event-based behaviours. In that way, this module could periodically check the control system's status, calculating if required a new set of control coefficients to be set to the *rCSP* processor.

- Research the role that a *Σ-Δ* modulation module could have in the performance, size and power consumption of the *rCSP* processor.
• Extension of the provided adaptation and filtering libraries included for the system-on-chip control solution.

• Inclusion of Watchdog functionalities in the system-on-chip architecture to provide a higher-level of reliability in case that more complex software routines are implemented.

• Development of the Client-Server architecture described in Section 7.8, which allows to obtain remotely the implementation-ready bitstream of the rCSP processor for any desired configuration.

8.6 Summary

This section concluded this thesis by summarising its objectives, achievements and how they have been reached, outlining at the end a number of potential extensions of this research.
Appendix A

FPGA devices

This appendix briefly reviews the main characteristics of a selected number of FPGAs that were used in the implementation of the rCSP processor and the system-on-chip solution.

Table A.1 lists the selected FPGA vendors and technologies, while the main parameters of these devices are summarised in Table A.2 (Excalibur), in Table A.3 (Spartan-IIE family) and Table A.4 (Virtex-II).

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Technology</th>
<th>Part</th>
<th>Package</th>
<th>Speed</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera</td>
<td>Excalibur ARM</td>
<td>EPXA10</td>
<td>FC1020</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>Xilinx</td>
<td>Spartan-IIE</td>
<td>XC2S200E</td>
<td>FG456</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>Xilinx</td>
<td>Spartan-IIE</td>
<td>XC2S600E</td>
<td>FG456</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>Xilinx</td>
<td>Virtex-II</td>
<td>XCV1000</td>
<td>BG575</td>
<td>-6</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Targeted FPGAs

ARM-Based Excalibur

Figure A.1 shows the structure of the Excalibur EPXA10 device while Table A.2 summarises the main logic parameters of this FPGA. This device embeds an ARM922T hard processor core that interfaces with the PLD (programmable logic device) section through a built-in AMBA bus. The embedded stripe contains the processor core, peripherals and a memory subsystem.
Appendix A: Implementation FPGAs. Parameters Summary

---

<table>
<thead>
<tr>
<th>Technology</th>
<th>Part Family</th>
<th>Logic Elements (LEs)</th>
<th>Embedded System Blocks</th>
<th>Maximum RAM Bits (EBS bytes)</th>
<th>Typical Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM-Based Excalibur</td>
<td>EPXA10 APEX 20KE</td>
<td>38,400</td>
<td>160</td>
<td>327,680</td>
<td>1,772,000</td>
</tr>
</tbody>
</table>

Table A.2: Excalibur FPGA main complexity parameters

Logic elements (LEs) are basic building blocks of the APEX 20K PLD implemented in the Excalibur device. A logic element consists of a look-up table (LUT) (i.e. a function generator that quickly computes any function of four variables) and a programmable register to support sequential functions. Some LEs feed output or bidirectional I/O pins on the device.

![Figure A.1: Excalibur Embedded Processor PLD Architecture [2]](image)

Spartan-IIE

The Spartan-IIE family of FPGAs is targeted towards very low cost solutions that require a good level of performance. These devices provide system clock rates beyond 200 MHz and they are implemented on a 0.15 micron technology. A Spartan-IIE FPGA is composed of five major configurable elements [49](see Figure A.2):

- IOBs (Input/Output Blocks), which provide the interface between the package pins and the internal logic.
- CLBs (Configurable Logic Blocks), which provide the functional elements for constructing most logic.
- Dedicated block RAM memories of 4096 bits (4 Kbytes) each.
Appendix A: Implementation FPGAs. Parameters Summary

- Clock DLLs (Delay-Locked Loops) for clock-distribution delay compensation and clock domain control.

- Versatile multi-level interconnect structure.

![Figure A.2: Basic Spartan-IIE Family FPGA Block Diagram [49]](image)

The basic building block of the CLB is the logic cell (LC). An LC includes a 4-input function generator (implemented as 4-input LUT), carry logic, and storage element. The output from the function generator in each LC drives the CLB output or the D input of the flip-flop. Each CLB contains four LCs, organised in two similar slices.

<table>
<thead>
<tr>
<th>Part</th>
<th>Logic Cells</th>
<th>Gate Range</th>
<th>Array (R x C)</th>
<th>Total CLBs</th>
<th>Available User I/O</th>
<th>Distributed RAM Bits</th>
<th>Block RAM Bits</th>
<th>Block Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2S200E</td>
<td>5,292</td>
<td>71,000 - 200,000</td>
<td>28 x 42</td>
<td>1,176</td>
<td>285</td>
<td>75,264</td>
<td>56K</td>
<td>14</td>
</tr>
<tr>
<td>XC2S600E</td>
<td>15,552</td>
<td>210,000 - 600,000</td>
<td>48 x 72</td>
<td>3,456</td>
<td>325</td>
<td>221,184</td>
<td>288K</td>
<td>72</td>
</tr>
</tbody>
</table>

Table A.3: Selected Spartan-IIE FPGAs main complexity parameters
Virtex-II

The Virtex-II family is a platform FPGA developed for high performance designs that are based on IP cores and customised modules. The family is often used in telecommunication, wireless, networking, video, and DSP applications. The 0.15 microns / 0.12 microns CMOS 8-layer metal process and the Virtex-II architecture are optimised for high speed with low power consumption. The main parameters of the Virtex-II device use for implementing the rCSP are summarised in Table A.4.

<table>
<thead>
<tr>
<th>Part</th>
<th>System Array (R x C)</th>
<th>CLB Slices</th>
<th>Distributed RAM Kbits</th>
<th>Multiplier 18 Kbits Blocks</th>
<th>Max RAM Blocks Kbits</th>
<th>GCLKs</th>
<th>IO Pads</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2V1000</td>
<td>1M 40 x 32</td>
<td>5,120</td>
<td>160</td>
<td>40</td>
<td>40</td>
<td>720</td>
<td>8 328</td>
</tr>
</tbody>
</table>

Table A.4: Virtex2 FPGA main complexity parameters

Its internal configurable logic includes four major elements organised in a regular array [51](see Figure A.3):

- CLBs, which provide functional elements for combinatorial and synchronous logic, including basic storage elements.

- BUFTs (3-state buffers) associated with each CLB element, which drive dedicated segmentable horizontal routing resources.

- Block SelectRAM memory modules, which provide large 18 Kbit storage elements of dual-port RAM.

- DCM (Digital Clock Manager) blocks.

CLB resources include four slices and two 3-state buffers. Each slice is equivalent and contains: a) Two function generators; b) Two storage elements; c) Arithmetic logic gates; d) Large multiplexors; e) Wide function capability; f) Fast carry look-ahead chain; g) Horizontal cascade chain (OR gate).

The function generators are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory. In addition, the two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches.
Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources. The block SelectRAM memory resources are 18 Kb of dual-port RAM, programmable from 16K x 1 bit to 512 x 36 bits, in various depth and width configurations. Each port is totally synchronous and independent, offering three "read-during-write" modes. Block SelectRAM memory is cascadable to implement large embedded storage blocks. Supported memory configurations for dual-port and single-port modes. A multiplier block is associated with each SelectRAM memory block. The multiplier block is a dedicated 18 x 18-bit multiplier and is optimized for operations based on the block SelectRAM content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM resource. Read/multiply/accumulate operations and DSP filter structures are extremely efficient. Both the SelectRAM memory and the multiplier resource are connected to four switch matrices to access the general routing resources.
Appendix B

rCSP's VHDL Hierarchy

The VHDL model of the rCSP processor is summarised in Table B.1, and the packages used in the model are presented in Table B.2. Note that only the top level entities of each module are listed.

<table>
<thead>
<tr>
<th>Entity</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcserv3</td>
<td>rcserv3.vhd</td>
<td>rCSP top level entity with APB slave interface</td>
</tr>
<tr>
<td>rcserv3_ahb</td>
<td>rcserv3_ahb.vhd</td>
<td>rCSP top level entity with AHB slave interface</td>
</tr>
<tr>
<td>apb_siface</td>
<td>apb_siface.vhd</td>
<td>APB Slave Interface</td>
</tr>
<tr>
<td>ahh_siface</td>
<td>ahh_siface.vhd</td>
<td>AHB Slave Interface</td>
</tr>
<tr>
<td>rcservcore</td>
<td>rcservcore.vhd</td>
<td>rCSP processor core</td>
</tr>
<tr>
<td>cl_mod</td>
<td>cl_mod.vhd</td>
<td>Control Law Module with integrated PC</td>
</tr>
<tr>
<td>ih</td>
<td>ih.vhd</td>
<td>Instruction Handler Module</td>
</tr>
<tr>
<td>il</td>
<td>il.vhd</td>
<td>Interlocking Module</td>
</tr>
<tr>
<td>alu</td>
<td>alu.vhd</td>
<td>Arithmetic Unit Module</td>
</tr>
<tr>
<td>mmu</td>
<td>mmu.vhd</td>
<td>Memory Management Module</td>
</tr>
<tr>
<td>cff_mod</td>
<td>cff_mod.vhd</td>
<td>Coefficient Module</td>
</tr>
<tr>
<td>stv_mod_apb</td>
<td>stv_mod_apb.vhd</td>
<td>State Variables Module</td>
</tr>
<tr>
<td>tgt_mod</td>
<td>tgt_mod.vhd</td>
<td>Target Module</td>
</tr>
<tr>
<td>reg_file</td>
<td>reg_file.vhd</td>
<td>Register File</td>
</tr>
<tr>
<td>d6b_loop</td>
<td>d6b_loop.vhd</td>
<td>Control Loop Tracker</td>
</tr>
<tr>
<td>o_mod</td>
<td>o_mod.vhd</td>
<td>Output Module</td>
</tr>
<tr>
<td>l_mod</td>
<td>l_mod.vhd</td>
<td>Input Module (without SAR units)</td>
</tr>
<tr>
<td>l_mod_sac</td>
<td>l_mod_sac.vhd</td>
<td>Input Module (with SAR units)</td>
</tr>
</tbody>
</table>

Table B.1: rCSP processor VHDL model hierarchy
### Table B.2: Packages used in the rCSP processor model

<table>
<thead>
<tr>
<th>Package</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resp_comp</td>
<td>rcsp_comp.vhd</td>
<td>rCSP's core components declaration</td>
</tr>
<tr>
<td>amba</td>
<td>amba.vhd</td>
<td>Type definitions for the AMBA bus</td>
</tr>
<tr>
<td>rcsp_core_config</td>
<td>rcsp_core_config.vhd</td>
<td>rCSP's core configuration file</td>
</tr>
<tr>
<td>rcsp_core_ifaces</td>
<td>rcsp_core_ifaces.vhd</td>
<td>Type and function declaration for module interfacing</td>
</tr>
<tr>
<td>rcsp_on_chip_config</td>
<td>rcsp_on_chip_config.vhd</td>
<td>On-chip configuration and type declaration parameters</td>
</tr>
</tbody>
</table>
Appendix C

ANSI-C Libraries

This appendix summarises the main device driver and control libraries developed for the system-on-chip solution.

Header: cntl_data_filters.h

int Average(int new_sensor_value, int nominal_ratio)

This function performs a window averaging of the data periodically passed through the parameter int new_sensor_value, thus acting as a low-pass filter. Detailed information about average window filtering can be found in [10]. The parameter int nominal_ratio is used as default filter output while the window is still being filled.

int Average_Int(int new_sensor_value, int nominal_ratio)

This function is equivalent to the previous one, but in order to minimise the arithmetic operations involved in the averaging process, it only accepts window sizes that are powers of two. The use of this function further reduces the embedded CPU requirements and shortens the overall adaptation time.

Constant ADPT_WINDOW_SIZE

This constant represents the number of elements that comprises the filtering window. Since the adaptation process is called periodically at intervals of $T_{adapt}$ seconds, the values will be averaged over a window of $T_{adapt} \times \text{ADPT\_WINDOW\_SIZE}$
seconds. In case of using the `Average_Int` function, the value assigned to this constant should be a power of 2.

- **Header:** `cntl_adaptation.h`
  
  `void Adaptation_Method_1(void)`
  
  This function implements a gain scheduling adaptation routine. First the ratio of two sensors' reading is calculated and filtered over a window of `ADPT_WINDOW_SIZE` elements. Then, the filtered ratio is used to select the proper set of coefficients to be sent to the rCSP. This function can be used as a template to build up different adaptation methods.

- **Header:** `rcspv3_mem.h`
  
  `void Write_CFF_Set(unsigned int set)`
  
  Write the selected set of coefficients into the rCSP.

  `unsigned int Read_Single_STV(unsigned int addr)`
  
  Retrieves a single state variable from the rCSP.

  `void Console_All_STV(void)`
  
  Sends all the state variables stored on the rCSP to the UART.

  `struct IRAM_Reading Read_Dual_Sensors(struct IRAM_Request iram_req)`
  
  Retrieves simultaneously two sensor readings from the rCSP.

  `unsigned int Read_Sensor(unsigned int sensor_nr)`
  
  Retrieves the reading of the selected sensor from the rCSP.

  `void Clear_All_Data(void)`
  
  Erases the all the data from the different rCSP modules.

  ```c
  struct IRAM_Reading {
    unsigned int sensor_1; // Sensor 1 Reading
    unsigned int sensor_2; // Sensor 2 Reading
  };
  ```
The structure `lRAM_Reading` is returned when two sensors are read simultaneously.

```c
struct lRAM_Request {
    unsigned int nr_sensor_1; // Sensor 1 Index
    unsigned int nr_sensor_2; // Sensor 2 Index
};
```

The structure `lRAM_Request` is passed as argument in order to read two sensors simultaneously.

- **Header:** `rcspv3.boot.h`

```c
void Boot_Algorithm(void) {
    Transfers the algorithms to be executed in the rCSP processor.
}

void Boot_Coefficients(void) {
    Transfers the initial set of coefficients into the rCSP processor.
}
```

- **Header:** `cpu_irq.h`

```c
void setup_adaptation_time(unsigned int time_ns) {
    Defines the adaptation period in nanoseconds.

    \[ \text{time}_{\text{ns}} = \frac{1}{\text{Adaptation frequency (Hz)}} \cdot 10^{-9} \]
```
Appendix D

Matlab functions

This appendix presents a number of useful functions developed in the thesis to obtain transfer functions and state-space representations in the delta form. Also, the main functions developed for generating and analysing rCSP's coefficients and state variables are summarised. The appendix finishes describing the rCSP Development Suite installation process.

D.1 Delta-Transform related functions

function [numdel,dendel] = tfz2tfdel(numz,denz,Ts)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numz,denz</td>
<td>Numerator and denominator coefficients of $F(z)$</td>
</tr>
<tr>
<td>Ts</td>
<td>Sampling Period (To be omit for Loughborough's $\gamma$)</td>
</tr>
<tr>
<td>numdel,dendel</td>
<td>Numerator and denominator coefficients of $F(\gamma)$</td>
</tr>
</tbody>
</table>

Table D.1: tfz2tfdel function's parameters list

This function transforms a transfer function in $z$ into a transfer function in $\gamma$. If the sampling period ($T_s$) is omitted or made equal to 1, the Loughborough definition of $\gamma$ is used (i.e. $\gamma = z - 1$). The numerator and denominator of $F(z)$ should be expressed in descending (positive) powers of $z$, as presented in Equation (D.1),

$$F(z) = \frac{b_n z^n + b_{n-1} z^{(n-1)} + \ldots + b_1 z + b_0}{a_n z^n + a_{n-1} z^{(n-1)} + \ldots + a_1 z + a_0}$$  \hspace{1cm} (D.1)
and in the same way, the output function $F(\gamma)$ will be returned in descending (positive) powers of $\gamma$, as shown in Equation (D.2).

$$F(\gamma) = \frac{b_n \gamma^n + b_{n-1} \gamma^{(n-1)} + ... + b_1 \gamma + b_0}{a_n \gamma^n + a_{n-1} \gamma^{(n-1)} + ... + a_1 \gamma + a_0}$$  \hspace{1cm} (D.2)

The transformation consist on replacing the $z^i$ terms in $F(z)$ with $(\gamma + 1)^i$ and calculate the new set of coefficients resulting from the polynomial expansion.

**function** [amod,bmod,cmod,dmod,p,q] = tfdel2ssmdel(numdel,dendel)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numdel,dendel</td>
<td>Numerator and denominator coefficients of $F(\gamma)$</td>
</tr>
<tr>
<td>amod,bmod,cmod,dmod</td>
<td>State space representation in the modified canonic delta form</td>
</tr>
</tbody>
</table>

Table D.2: Function tfdel2ssmdel parameters list

This function, generates the state space representation in the modified canonic delta form from a transfer function $F(\gamma)$. The numerator and denominator of $F(\gamma)$ should be expressed in descending (positive) powers of $\gamma$, as in Equation (D.2). The structure of the matrices returned is presented below:

$$
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_{n + 1}
\end{pmatrix}
= 
\begin{pmatrix}
    1 & a_1 & 0 & \ldots & 0 \\
    0 & 1 & a_2 & \ddots & 0 \\
    \vdots & \vdots & \ddots & \ddots & \vdots \\
    0 & \ldots & 0 & 1 & a_{n-1} \\
    -a_n & -a_n & \ldots & -a_n & (1 - a_n)
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n \\
\end{pmatrix}
+ 
\begin{pmatrix}
    0 \\
    0 \\
    \vdots \\
    a_n
\end{pmatrix}
$$  \hspace{1cm} (D.3)

$$y_k = \begin{pmatrix}
    c_1 - c_0 |c_2 - c_0| \ldots |c_n - c_0 |
\end{pmatrix}
+ d \cdot u_k$$  \hspace{1cm} (D.4)

**function** [Amod,Bmod,Cmod,Dmod]=tfs2ssmdel(numc,denc,Ts,method,delta_type)

This function discretises the continuous transfer function $F(s)$ and generates its modified canonic delta state space representation. The numerator and denominator of
Appendix D: Delta related Matlab functions

Table D.3: Function tfs2ssmdel parameters list

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numdc,dendc</td>
<td>Numerator and denominator coefficients of $F(s)$</td>
</tr>
<tr>
<td>$T_s$</td>
<td>Sampling Period (Compulsory)</td>
</tr>
<tr>
<td>method</td>
<td>Discretisation method ('zoh', 'tustin', 'matched')</td>
</tr>
<tr>
<td>deltatype</td>
<td>'iboro' for $\gamma = z - 1$</td>
</tr>
<tr>
<td></td>
<td>'gwlin' for $\gamma = (z - 1) / T_s$ if omitted, $\gamma = z - 1$ is assumed</td>
</tr>
<tr>
<td>Amod,Amod,Cmod,Dmod</td>
<td>Discrete state space representation in the modified canonic delta form</td>
</tr>
</tbody>
</table>

$F(s)$ should be expressed in descending (positive) powers of $s$, as shown in Equation (D.5).

$$F(s) = \frac{b_n s^n + b_{n-1} s^{n-1} + ... + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + ... + a_1 s + a_0} \quad (D.5)$$

The process involves three different stages:

1. $F(s)$ to $F(z)$ transformation
   Using the standard MATLAB function c2dm, the continuous transfer function $F(s)$ is discretised (using the selected discretisation method) at the requested sampling period ($T_s$). The output is a transfer function $F(z)$ in descending (positive) powers of $z$.

2. $F(z)$ to $F(\gamma)$ transformation
   The function tfz2tfdel is used to transform $F(z)$ into $F(\gamma)$.

3. $F(\gamma)$ to State Space Representation (A,B,C,D)
   The function tfdel2ssmdel is used to generate the state space representation of $F(\gamma)$ in the modified delta canonic form.

function [nums,dens,orders,nr_in,nr_out]=ssz2tfdel(sys)

Table D.4: Function ssz2tfdel parameters list

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys</td>
<td>MATLAB's representation (using the ss command) of a Z-Transform-based state-space system</td>
</tr>
<tr>
<td>nums,dens</td>
<td>Array of numerators and denominators from the generated transfer functions in delta</td>
</tr>
<tr>
<td>orders</td>
<td>Order of the generated transfer functions in delta</td>
</tr>
<tr>
<td>nr_in</td>
<td>Number of system inputs</td>
</tr>
<tr>
<td>nr_out</td>
<td>Number of system outputs</td>
</tr>
</tbody>
</table>
Appendix D: Delta related Matlab functions

This function transforms the Z-Transform-based state-space representation of a MIMO system (i.e. \( A_z, B_z, C_z, D_z \)) into its equivalent array of delta-formulated SISO transfer functions \( F_{ij}(\gamma) \).

D.2 Functions to work with rCSP’s coefficients and state variables

function \([b\_str,\text{mantissa},\text{exponent},\text{conv\_error}] = \text{num2cff}(\text{varargin})\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varargin{1}</td>
<td>Array of real numbers to be converted</td>
</tr>
<tr>
<td>varargin{2}</td>
<td>Coefficient parameters (optional) [mantissa,exponent,bias]</td>
</tr>
<tr>
<td>b_str</td>
<td>Array of converted coefficients in binary string format</td>
</tr>
<tr>
<td>mantissa</td>
<td>Array containing the coefficients’ mantissa</td>
</tr>
<tr>
<td>exponent</td>
<td>Array containing the coefficients’ exponent</td>
</tr>
<tr>
<td>conv_error</td>
<td>Array of quantisation errors</td>
</tr>
</tbody>
</table>

Table D.5: Function tfde12ssmdel parameters list

This function converts an array of real numbers into an array of rCSP’s floating-point coefficients. The input parameters and returned values are summarised in Table D.2. Note that the parameters bias can be omitted. Note that the when the coefficient’s parameters are omitted the standard rCSP configuration is assumed. If a configuration is provided and the bias is omitted, the default rCSP bias, as presented in Equation (4.47) is applied.

Usage example:

```matlab
>> [b\_str,mantissa,exponent,conv\_error] =
   num2cff([-1, 0, 0.34, 2, 1.25, 33],[6,6])

ans\_1 =
   b\_str=['111111111110';'000000000000';'010110111000'
   '010000111011';'010100111010';'010001111111']
   mantissa=[-1 0 22 16 20 17]
   exponent=[62 56 59 58 63]
   conv\_error=[0 0 0.0037 0 0 1]
```
function num_list = cff2num(varargin)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varargin{1}</td>
<td>Binary string array of rCSP's coefficients</td>
</tr>
<tr>
<td>varargin{2}</td>
<td>Coefficient parameters (optional) [mantissa,exponent,bias]</td>
</tr>
<tr>
<td>num_list</td>
<td>Array of rCSP's coefficients' real values</td>
</tr>
</tbody>
</table>

Table D.6: Function cff2num parameters list

This function accepts rCSP's floating-point coefficients represented in binary format and returns their real value. The input parameters and returned values are summarised in Table D.2. Note that when the coefficient's parameters are omitted, the standard rCSP configuration is assumed. If a configuration is provided and the bias is omitted, the default rCSP bias, as presented in Equation (4.47) is applied.

Usage example:

```matlab
>> num_list = cff2num(["010110111000'
    '010001111011'; '010100111010'
    '010001111111'
    ]
    ,[6,6])
Answer:
num_list = [2.0000 1.2500 34.0000]
```

function [quant_cff_out, conv_error_out] = quantcff(varargin)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varargin{1}</td>
<td>Array of real numbers to be converted</td>
</tr>
<tr>
<td>varargin{2}</td>
<td>Coefficient parameters (optional) [mantissa,exponent,bias]</td>
</tr>
<tr>
<td>quant_cff_out</td>
<td>Array of quantised coefficients (real number)</td>
</tr>
<tr>
<td>conv_error_out</td>
<td>Array of quantisation error</td>
</tr>
</tbody>
</table>

Table D.7: Function quantcff parameters list

This function accepts real numbers and returns their quantised value and error for the specified coefficient's parameters. The input parameters and returned values are summarised in Table D.2. Note that when the coefficient's parameters are omitted, the standard rCSP configuration is assumed. If a configuration is provided and the bias is omitted, the default rCSP bias, as presented in Equation (4.47) is applied.

Usage example:
Appendix D: Delta related Matlab functions

>> [quantquant_cff_out, conv_error_out]=
    quantcff([0.00125 0.99875 1 10.35 40])
Answer =
quantquant_cff_out = [0.0012 1.0000 1.0000 10.6000 40.0000]
conv_error_out = [-0.0000 0.0012 0 0.1500 0]

function [stv_bin_list,quant_error] = num2stv(varargin)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varargin{1}</td>
<td>Array of real numbers to be converted</td>
</tr>
<tr>
<td>varargin{2}</td>
<td>Calculate quantisation error (optional) (true,false)</td>
</tr>
<tr>
<td>varargin{3}</td>
<td>State Variables parameters (optional) [integer bits,frac bits]</td>
</tr>
<tr>
<td>stv_bin_list</td>
<td>Array of converted state variables in binary string format</td>
</tr>
<tr>
<td>quant_error</td>
<td>Array of quantisation errors</td>
</tr>
</tbody>
</table>

Table D.8: Function num2stv parameters list

This function converts an array of real numbers into an array of State Variables in rCSP’s fixed point format. The input parameters and returned values are summarised in Table D.2. Note that the when the state variable's parameters are omitted the standard rCSP configuration is assumed.

Usage example:

>> [stv_bin_list,quant_error] = num2stv([-1.1 127.32],1,[12,12])
Answer =
stv_bin_list = ['11111111111111111111111111111111';
                  '00000000000000000000000000000000']
quant_error = 1.0e-004 + [ 0.9766 -0.6836]

function [num] = stv2num( varargin )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varargin{1}</td>
<td>Binary string array of state variables to be converted</td>
</tr>
<tr>
<td>varargin{2}</td>
<td>State Variables parameters (optional) [integer bits,frac bits]</td>
</tr>
<tr>
<td>num</td>
<td>Array of converted state variables (real value)</td>
</tr>
</tbody>
</table>

Table D.9: Function stv2num parameters list

This function converts an array of rCSP’s fixed-point state variables into an array whose elements contain the real values of these state variables. The input parameters
and returned values are summarised in Table D.2. Note that the when the state variable’s parameters are omitted the standard rCSP configuration is assumed.

Usage example:

```matlab
>> [ num] = stv2num(['111111111110110011011';
    '000000011111101001000111111'])

Answer =
Assuming rCsp v3.0 standard configuration
num = [-1.100 127.3201]
D.3 Installation and Execution of the \textit{rCSP Development Suite}

In order to install the Matlab-based \textit{rCSP Development Suite}, the user should copy or unzip its directory structure on the desired location in the hard-drive. This location must be made available to the software tool by creating an entry in the operating system's environment list. If /home/usr/rcspv30 were the installation directory, the environment variable should be defined as follows:

\begin{verbatim}
rcspv3_matlab_mpe = /home/usr/rcspv30
\end{verbatim}

Once the environment list has been updated, the \textit{rCSP Development Suite} can be launched by executing (in Matlab) the following command:

\begin{verbatim}
>> start_rcspv3_env
\end{verbatim}

This scripts modifies MATLAB's environment and path structure and starts the Application Launcher tool bar (see Figure D.1) which provides direct access to the different applications needed throughout the design, test and implementation of an algorithm in the rCSP. The toolbar can be also manually open with the command:

\begin{verbatim}
>> launcher
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{application_launcher.png}
\caption{Application Launcher}
\end{figure}
Appendix E

Mathematical Operators

E.1 Continuous-Time Operators

- Derivative Operator: \( \rho = \frac{d}{dt} \)

\[ \rho(f(t)) \equiv \rho f(t) \equiv \frac{df(t)}{dt} \]

This operator, when applied recursively to the same function has the following notation and value:

\[ \rho^n(f(t)) \equiv \rho^n f(t) \equiv \frac{d^n f(t)}{dt^n} \quad \forall n \in \mathbb{Z} \quad (E.1) \]

E.2 Discrete-Time Operators

This section assumes uniform sampling period. A sampled signal \( f(t) \) is represented as a sequence of values \( f[kT] \) where \( T \) represents the sampling period and \( k \) takes zero or positive integers. For convenience, \( T \) will be in most cases omitted.

- Forward Shift Operator: \( q \)

\[ q(f(t)) \equiv q(f[k]) \equiv qf[k] \equiv f[k + 1] \quad \forall k \in \mathbb{Z} \quad (E.2) \]

This operator, when applied recursively to the same function has the following notation and value:

\[ q^n f[k] = f[k + n] \quad \forall n \in \mathbb{Z} \quad (E.3) \]
Appendix E: Mathematical Operators

- **Delay Operator:** $q^{-1}$

  $q^{-1}(f(t)) = q^{-1}(f[k]) = q^{-1}f[k] = f[k-1] \quad \forall k \in Z^+$ \hspace{1cm} (E.4)

  and applied recursively:

  $q^{-n}f[k] = f[k-n] \quad \forall n \in Z \hspace{1cm} (E.5)$

- **Delta Operator (Classic definition):** $\delta_{\text{CLASS}}$

  $\delta_{\text{CLASS}}(f(t)) \equiv \delta_{\text{CLASS}}f[k] \equiv \frac{f(k+1) - f(k)}{T} \hspace{1cm} (E.6)$

  and applied recursively:

  $\delta^n_{\text{CLASS}}f[k] = \frac{1}{T^n} \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f[k+i] \quad \forall n \in Z \hspace{1cm} (E.7)$

  The demonstration is similar to the one shown for Equation E.9.

- **Delta Operator (Loughborough definition):** $\delta$

  $\delta(f(t)) \equiv \delta f[k] \equiv f(k+1) - f(k) \hspace{1cm} (E.8)$

  and applied recursively:

  $\delta^n f[k] = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f[k+i] \quad \forall n \in Z \hspace{1cm} (E.9)$

  Demonstration:

  \[
  \begin{align*}
  \delta f[k] & = f[k+1] - f[k] \\
  \delta^2 f[k] & = \delta(f[k+1] - f[k]) = f[k+3] - 2f[k+1] + f[k] \\
  \delta^3 f[k] & = f[k+3] - 3f[k+2] + 3f[k+1] - f[k] \\
  & \vdots \\
  \delta^n f[k] & = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f[k+i]
  \end{align*}
  \]

- **Inverse Delta Operator (Loughborough definition):** $\delta^{-1}$

  In the same way that the $\delta$ operator represents a differentiation process, its inverse represents the accumulation process depicted in Figure E.1.

  $\delta^{-1}u(k) = y[k-1] + u[k-1] \hspace{1cm} (E.10)$
Demonstration:
By inspecting Figure E.1 two immediate relations can be found:

\[ y[k] = y[k-1] + u[k-1] \]  \hspace{1cm} (E.11)

\[ \delta^{-1}u(k) = y[k] \]  \hspace{1cm} (E.12)

Then, substituting Equation (E.11) into Equation (E.12) leads to Equation (E.10).

\[ \delta=y[k] \]

\[ \delta^{-1}u(k) = y[k] \]

\[ \delta = T \cdot \delta \text{CLASS} \]  \hspace{1cm} (E.15)

\[ \delta = q - 1 \]  \hspace{1cm} (E.16)

\[ q = \delta + 1 \]  \hspace{1cm} (E.17)
Appendix F

Transform Techniques in Control

All the transforms included in this appendix posses common properties like linearity, shift theorem, final value theorem, etc., that can be extensively found in control and mathematic literature (e.g. [41], [39], [9]). Therefore, this appendix only summarises their definition and the relation between their operators and variables. Afterwards, different discretisation methods are presented that lead to a transfer function in $\gamma$. This appendix also demonstrates several statements made in Chapter 4 regarding discrete systems expressed in the delta form.

F.1 Continuous-Time transforms

F.1.1 Laplace Transform ($L[f(t)]$)

Let be $f(t)$ a function of time $t$. In many real problems only values of $t \geq 0$ are of interest. Hence $f(t)$ is given for $t \geq 0$, and for all $t < 0$, $f(t)$ is taken to be 0. Then, the Laplace transform of $f(t)$ is defined as:

$$L[y(t)] \equiv Y(s) = \int_{0}^{\infty} e^{-st}y(t)dt$$ (F.1)

The Laplace transform maps, or transforms, the function $f(t)$ into a different function $F(s)$, which is a function of the complex variable $s$. This transforms has the property of transforming linear differential equations into algebraic equations, thus
being extremely useful in control.

**F.2 Discrete-Time transforms**

### F.2.1 Z-Transform

The traditional discrete equivalent of the Laplace transform of a function is the Z-transform of a sequence [39].

The (one-sided) Z-Transform of a time function $x(t)$ or of a sequence of values $x(kT)$ is defined by the following equation:

$$X(z) \equiv Z\{x(t)\} \equiv Z\{x(kT)\} = \sum_{k=0}^{\infty} x(kT)z^{-k}, \quad z = e^{sT} \quad (F.2)$$

where $t$ is nonnegative and $T$ is the sampling period.

The shifting theorem states that:

$$Z\{q^n x[k]\} = z^n X(z) - \left( \sum_{k=0}^{n-1} x[k] z^{n-k} \right) \quad (F.3)$$

$$Z\{q^{-n} x[k]\} = z^{-n} X(z) \quad (F.4)$$

The previous equations show that the $q$ operator is linked with the Z-Transform variable $z$.

The Z-transform has the property of transforming discrete series of values and difference equations into algebraic equations. Therefore it is widely used in discrete mathematics and signal processing. Digital systems take advantage of the resemblance between the inverse of its transformation variable $z^{-1}$ and the delay that takes place while transferring data between two register. This highly intuitive design characteristic, comes at the cost of word length. At high-sample rates, excessive precision is needed in order to store different coefficients and variables. This wide dynamic range is due to the fact the transform is calculated using the value of the signal at each sample, and since these values tend to get more and more similar as the sample rate increases, more and more precision is required to differentiate them.
F.2.2 Delta-Transform

The Delta-transform is based on the $\delta$ operator, and therefore it processes the differences between two consecutive sample values, rather than their natural value. In that way the dynamic range is reduced, allowing designs with shorter word length, and therefore producing smaller and faster implementations.

The Delta-transform can be defined in terms of the Z-transform as:

$$X(\gamma) \equiv D\{x(t)\} = Z\{x(t)\} \bigg|_{z=1+\gamma} \quad (F.5)$$

Thus, the relation between both transform variables is:

$$\gamma = z - 1 \quad (F.6)$$

This leads to the definition of the Delta-transform of a time function $x(t)$, where $t$ is nonnegative, or of a sequence of values $x(kT)$, where $k$ takes zero or positive integers and $T$ is the sampling period, as:

$$X(\gamma) \equiv D\{x(t)\} \equiv D\{x[kT]\} = \sum_{k=0}^{\infty} x(k) \cdot (1 + \gamma)^{-k} \quad (F.7)$$

The shift theorem states that:

$$D\{\delta x[k]\} = \gamma X(\gamma) - x[0](1 + \gamma) \quad (F.8)$$
$$D\{\delta^{-1} x[k]\} = \gamma^{-1} X(\gamma) \quad (F.9)$$

which shows that the $\delta$ operator and the Delta-transform variable $\gamma$ are linked in the same way that $z$ and $q$ are.

F.3 Discretising by variable's substitution

There is a number of methods available for discretising continuous transfer functions just by approximating the $s$ variable, for instance:

1. Euler's method (forward rectangular area)

$$s \approx \frac{z - 1}{T} \quad (F.10)$$
2. Backward rectangular area

\[ s \approx \frac{z - 1}{T \cdot z} \]  

(F.11)

3. Tustin's method (Bilinear Transformation or trapezoidal rule)

\[ s \approx \frac{2(z - 1)}{T(z + 1)} \]  

(F.12)

This mapping can be seen as an expansion of \( z \) by a Taylor's series of first order as follows:

\[ z = e^{sT} = \frac{e^{sT/2}}{e^{-sT/2}} \approx \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}} \]  

(F.13)

Then, reordering terms in \( s \) the Tustin mapping between \( s \) and \( \gamma \) is obtained by substituting \( z \) for \( \gamma + 1 \):

\[ s \approx \frac{2\gamma}{T(\gamma + 2)} \]  

(F.14)

Figure F.1, depicts different routes that lead to a discretised transfer function \( F(\gamma) \) from a continuous function \( F(s) \). It is important to notice that a zero-order-hold does not imply any approximation.

![](image)

Figure F.1: Common discretising routes to obtain \( F(\gamma) \)

### F.3.1 From \( F(z) \) to \( F(\gamma) \)

Let \( F(z) \) be a general discrete transfer function in \( z \) defined as:

\[ F(z) = \frac{b_n z^n + b_{n-1} z^{n-1} + \ldots + b_1 z + b_0}{a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0} \]  

(F.15)
Equation F.6 showed the relation between the transform variables \( z \) and \( \gamma \). In order to transform \( F(z) \) into \( F(\gamma) \), \( z \) is substituted by \( \gamma + 1 \), obtaining the following transfer function:

\[
F(z) = \frac{b_n(\gamma + 1)^n + b_{n-1}(\gamma + 1)^{(n-1)} + \ldots + b_1(\gamma + 1) + b_0}{a_n(\gamma + 1)^n + a_{n-1}(\gamma + 1)^{(n-1)} + \ldots + a_1(\gamma + 1) + a_0}
\]  

(F.16)

The polynomial expansion of \( (\gamma + 1)^n \) can be calculated as:

\[
(\gamma + 1)^n = \binom{n}{0} \gamma^n + \binom{n}{1} \gamma^{n-1} + \ldots + \binom{n}{n-1} \gamma + 1
\]  

(F.17)

Substituting (F.17) separately in the numerator and denominator of (F.16), and rearranging terms in \( \gamma \) it is obtained:

\[
\text{num}(\gamma) = b_n \left( \frac{n}{n} \right) \gamma^n + \sum_{i=0}^{n} b_{n-i} \left( \frac{i}{n-i} \right) \gamma^{n-1} + \ldots + \sum_{i=1}^{n} b_1 \left( \frac{i}{1} \right) \gamma + \sum_{i=0}^{n} b_0
\]

\[
\text{den}(\gamma) = a_n \left( \frac{n}{n} \right) \gamma^n + \sum_{i=0}^{n} a_{n-i} \left( \frac{i}{n-i} \right) \gamma^{n-1} + \ldots + \sum_{i=1}^{n} a_1 \left( \frac{i}{1} \right) \gamma + \sum_{i=0}^{n} a_0
\]

(F.18)

Finally the transfer function \( F(\gamma) \) in terms of \( F(z) \)’s coefficients is:

\[
F(\gamma) = \frac{b_n \gamma^n + b_{n-1} \gamma^{n-1} + \ldots + b_1 \gamma + b_0}{a_n \gamma^n + a_{n-1} \gamma^{n-1} + \ldots + a_1 \gamma + a_0}
\]  

(F.19)

where

\[
b'_k = \sum_{i=k}^{n} b_i \binom{i}{k}
\]

\[
a'_k = \sum_{i=k}^{n} a_i \binom{i}{k}
\]

(F.20)

F.3.2 From \( F(s) \) to \( F(\gamma) \)

Let \( F(s) \) be a general continuous transfer function in \( s \) defined as:

\[
F(s) = \frac{\beta_n s^n + \beta_{n-1}s^{(n-1)} + \ldots + \beta_1 s + \beta_0}{\alpha_n s^n + \alpha_{n-1}s^{(n-1)} + \ldots + \alpha_1 s + \alpha_0}
\]  

(F.21)

In order to transform \( F(s) \) directly into \( F(\gamma) \) the bilinear transform shown in Equation (F.14) is applied, obtaining the following transfer function:

\[
F(\gamma) = \frac{\beta_n \left( \frac{2\gamma}{(\gamma+1)} \right)^n + \beta_{n-1} \left( \frac{2\gamma}{(\gamma+1)} \right)^{(n-1)} + \ldots + \beta_1 \left( \frac{2\gamma}{(\gamma+1)} \right) + \beta_0}{\alpha_n \left( \frac{2\gamma}{(\gamma+1)} \right)^n + \alpha_{n-1} \left( \frac{2\gamma}{(\gamma+1)} \right)^{(n-1)} + \ldots + \alpha_1 \left( \frac{2\gamma}{(\gamma+1)} \right) + \alpha_0}
\]  

(F.22)
Operating, \( F(\gamma) \) becomes

\[
F(\gamma) = \frac{\beta_0 (2\gamma)^n + \beta_{-1} T (\gamma + 2)^{n-1} + \beta_{-2} T^2 (\gamma + 2)^{n-2} + \ldots + \beta_{T^n-1} (\gamma + 2)^{n-T^n-1} + \beta_{T^n} (\gamma + 2)^n 
}{\alpha_n (2\gamma)^n + \alpha_{n-1} T (\gamma + 2)^{n-1} + \alpha_{n-2} T^2 (\gamma + 2)^{n-2} + \ldots + \alpha_{T^n-1} (\gamma + 2)^{n-T^n-1} + \alpha_{T^n} (\gamma + 2)^n}
\]

The polynomial expansion of \((\gamma + 2)^n\) can be calculated as:

\[
(\gamma + 2)^n = \binom{n}{0}\gamma^n + \binom{n}{1}\gamma^{n-1}2 + \ldots + \binom{n}{n-1}\gamma^{2n-1} + \binom{n}{n}2^n \tag{F.24}
\]

Substituting (F.24) separately in the numerator and denominator of (F.23), and rearranging terms in \( \gamma \) it is obtained:

\[
F(\gamma) = \frac{\tilde{b}_n \gamma^n + \tilde{b}_{n-1} \gamma^{n-1} + \ldots + \tilde{b}_1 \gamma \tilde{b}_0}{\tilde{a}_n \gamma^n + \tilde{a}_{n-1} \gamma^{n-1} + \ldots + \tilde{a}_1 \gamma \tilde{a}_0} \tag{F.25}
\]

where

\[
\tilde{b}_{n-k} = \sum_{i=k}^{n} \beta_{n-i} T^i \binom{i}{k} 2^{n-i+k}, \quad k = 0, 1, \ldots, n \tag{F.26}
\]

\[
\tilde{a}_{n-k} = \sum_{i=k}^{n} \alpha_{n-i} T^i \binom{i}{k} 2^{n-i+k}, \quad k = 0, 1, \ldots, n \tag{F.27}
\]

In order to obtain \( F(\gamma) \) in terms of \( \gamma^{-1} \) the numerator and denominator of Equation (F.25) are divided by \( \gamma^n \), obtaining:

\[
F(\gamma) = \frac{b'_n \gamma^{-n} + b'_{n-1} \gamma^{-(n+1)} + \ldots + b'_1 \gamma^{-1} + b'_0}{a'_n \gamma^{-n} + a'_{n-1} \gamma^{-(n+1)} + \ldots + a'_1 \gamma^{-1} + a'_0} \tag{F.28}
\]

where \( b'_k \equiv \tilde{b}_{n-k} \) and therefore:

\[
b'_k = \sum_{i=k}^{n} \beta_{n-i} T^i \binom{i}{k} 2^{n-i+k}, \quad k = 0, 1, \ldots, n \tag{F.29}
\]

\[
a'_k = \sum_{i=k}^{n} \alpha_{n-i} T^i \binom{i}{k} 2^{n-i+k}, \quad k = 0, 1, \ldots, n \tag{F.30}
\]

Finally, numerator and denominator of Equation (F.28) are divided by \( a'_0 \), leading to:

\[
F(\gamma) = \frac{b_n \gamma^{-n} + b_{n-1} \gamma^{-(n+1)} + \ldots + b_1 \gamma^{-1} + b_0}{a_n \gamma^{-n} + a_{n-1} \gamma^{-(n+1)} + \ldots + a_1 \gamma^{-1} + 1} \tag{F.31}
\]

where

\[
b_k = \frac{b'_k}{a'_0}, \quad k = 0, 1, \ldots, n \tag{F.32}
\]

\[
a_k = \frac{a'_k}{a'_0}, \quad k = 0, 1, \ldots, n \tag{F.33}
\]

Equation (F.31) represents the transfer function in \( \gamma^{-1} \) introduced in Chapter 4 to obtain the state-space delta and modified delta canonic forms.
Appendix F: Transform Techniques in Control

F(\gamma) analysis at high-sample rates

As sample rates increase, \( T \to 0 \). Expanding Equation (F.32)

\[
 b_k = \frac{b_k'}{\alpha_0} = \frac{\sum_{i=k}^{n} \beta_{n-i} T^i (i) 2^{n-i+k}}{\sum_{i=0}^{n} \alpha_{n-i} T^i 2^{n-i}} \tag{F.34}
\]

This can be rewritten as:

\[
 b_k = \frac{\sigma_k T^k + \sigma_{k+1} T^{k+1} + \cdots + \sigma_n T^n}{\varrho_0 + \varrho_1 T + \varrho_2 T^2 + \cdots + \varrho_n T^n} \tag{F.35}
\]

Calculating the limit when \( T \to 0 \):

\[
 \lim_{T \to 0} b_k = \frac{0}{\varrho_0} = \frac{0}{\alpha_n 2^n} = 0, \quad k > 0 \tag{F.36}
\]

Clearly, this result is valid if \( \alpha_n \neq 0 \) and \( \alpha_n \) is not proportional to \( T \). For \( k = 0 \):

\[
 \lim_{T \to 0} b_0 = \frac{\sigma_0}{\varrho_0} = \frac{\beta_n}{\alpha_n} \tag{F.37}
\]

In a similar way, the limit value of the \( \alpha_k \) coefficients can be calculated as:

\[
 a_k = \frac{a_k'}{\alpha_0} = \frac{\sum_{i=k}^{n} \alpha_{n-i} T^i (i) 2^{n-i+k}}{\sum_{i=0}^{n} \alpha_{n-i} T^i 2^{n-i}} \tag{F.38}
\]

This can be rewritten as:

\[
 a_k = \frac{\sigma_k T^k + \sigma_{k+1} T^{k+1} + \cdots + \sigma_n T^n}{\varrho_0 + \varrho_1 T + \varrho_2 T^2 + \cdots + \varrho_n T^n} \tag{F.39}
\]

which in the limit when \( T \to 0 \):

\[
 \lim_{T \to 0} a_k = \frac{0}{\varrho_0} = \frac{0}{\alpha_n 2^n} = 0, \quad k > 0 \tag{F.40}
\]

Again, this result is valid if \( \alpha_n \neq 0 \) and \( \alpha_n \) is not proportional to \( T \).

These results demonstrate that the coefficients of a transfer function in \( \gamma^{-1} \) tend to move towards zero as the sample rates increase, except for \( a_0 = 1 \) and \( b_0 \) which move towards \( \frac{\varrho_0}{\alpha_n} \):

Modified canonic \( \gamma \) form analysis at high-sample rates

Section 4.2.2 presented the modified canonic \( \gamma \) form state-space representation of a transfer function \( F(\gamma) \) defined as Equation (F.31). In this section the modified delta
canonic coefficients $q$ and $p$ (as defined in Figures 4.2 and 4.3) are analysed as the sample time tends to zero. As these figures show,

$$q_{n-j+1} = \frac{a_j}{a_{j-1}}, \quad j = 1, \ldots, n$$  \hspace{1cm} (F.41)

$$p_{n-j+1} = \frac{b_j}{a_j}, \quad j = 1, \ldots, n$$  \hspace{1cm} (F.42)

$$p_0 = b_0$$  \hspace{1cm} (F.43)

The following analysis assumes that the set of coefficients $\{a_k, b_k\}$ are not proportional to $T$. Applying Equation (F.33):

$$q_{n-j+1} = \frac{a_j}{a_{j-1}} = \frac{\sum_{i=j}^{n} \alpha_{n-i} T^i(i^2)}{\sum_{i=j-1}^{n} \alpha_{n-i} T^i(j^2) 2^{n-i+j-1}}$$  \hspace{1cm} (F.44)

This can be rewritten as:

$$q_{n-j+1} = \frac{a_j}{a_{j-1}} = \frac{\sigma_j T^j + \sigma_{j+1} T^{j+1} + \ldots + \sigma_n T^n}{\theta_{j-1} T^{j-1} + \theta_j T^j + \theta_{j+1} T^{j+1} + \ldots + \theta_n T^n}$$  \hspace{1cm} (F.45)

Dividing numerator and denominator by $T^n$:

$$q_{n-j+1} = \frac{a_j}{a_{j-1}} = \frac{\sigma_j T^{j-n} + \sigma_{j+1} T^{j-n+1} + \ldots + \sigma_n T^n}{\theta_{j-1} T^{j-n-1} + \theta_j T^{j-n} + \theta_{j+1} T^{j-n+1} + \ldots + \theta_n T^n}$$  \hspace{1cm} (F.46)

Calculating the limit when $T \to 0$:

$$\lim_{T \to 0} q_{n-j+1} = \frac{\infty^{n-j} + \infty^{n-j-1} + \ldots}{\theta_{j-1} \infty^{n-j-1} + \infty^{n-j} + \infty^{n-j-1} + \ldots} = \frac{1}{\infty} = 0$$  \hspace{1cm} (F.47)

Therefore, the $q$ coefficients of the $A$ and $B$ matrices that define the state-space modified delta form as presented in Equation (4.38) tend to zero as the sample rates increases.

In order to analyse the $p$ coefficients, the same procedure is used:

$$p_{n-j+1} = \frac{b_j}{a_j} = \frac{\sum_{i=j}^{n} \beta_{n-i} T^i(i^2) 2^{n-i+j}}{\sum_{i=j}^{n} \alpha_{n-i} T^i(j^2) 2^{n-i+j}}$$  \hspace{1cm} (F.48)

This can be rewritten as:

$$p_{n-j+1} = \frac{b_j}{a_j} = \frac{\tau_j T^j + \tau_{j+1} T^{j+1} + \ldots + \tau_n T^n}{\sigma_j T^j + \sigma_{j+1} T^{j+1} + \ldots + \sigma_n T^n}$$  \hspace{1cm} (F.49)

Dividing numerator and denominator by $T^n$:

$$p_{n-j+1} = \frac{b_j}{a_j} = \frac{\tau_j T^{j-n} + \tau_{j+1} T^{j-n+1} + \ldots + \tau_n}{\sigma_j T^{j-n} + \sigma_{j+1} T^{j-n+1} + \ldots + \sigma_n}$$  \hspace{1cm} (F.50)
Calculating the limit when $T \to 0$:

$$
\lim_{T \to 0} p_{n-j+1} = \frac{\tau_j}{\sigma_j} = \frac{\beta_{n-j}}{\alpha_{n-j}}, \quad j = 1, \ldots, n
$$

(F.51)

Therefore, the limit of the $C$ and $D$ matrices presented in Equation (4.39) for the modified canonic delta form, as the sample rate increases has the following value:

$$
\lim_{T \to 0} C = \frac{\beta_0}{\alpha_0} - \frac{\beta_1}{\alpha_1} \left| \frac{\beta_n}{\alpha_n} - \cdots - \frac{\beta_{n-1}}{\alpha_{n-1}} \right|
$$

(F.52)

$$
\lim_{T \to 0} D = \lim_{T \to 0} p_0 = \frac{\beta_n}{\alpha_n}
$$

(F.53)

where $C = [p_1 - p_0 | p_2 - p_0 | \cdots | p_n - p_0]$. 
Appendix G

Publications

Conference Proceedings:


Journal Papers:

Dario L. Sancho-Pradel and Roger M. Goodall. Targeted Processing for Real-Time Embedded Mechatronic Systems. Paper selected for publication on: on the special issue of IFACs: Control Engineering Practice, Elsevier, 2005

Bibliography


[27] Texas Instruments, [www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm](http://www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm).


